

---

# CASL — the Common Algebraic Specification Language

Till Mossakowski<sup>1</sup>, Anne E. Haxthausen<sup>2</sup>, Donald Sannella<sup>3</sup>, and Andrezej Tarlecki<sup>4</sup>

<sup>1</sup> DFKI Lab Bremen and University of Bremen, DE-28334 Bremen (P.O.Box 330 440) Germany [till@tzi.de](mailto:till@tzi.de)

<sup>2</sup> Department of Informatics and Mathematical Modelling, Technical University of Denmark, DK-2800 Kgs. Lyngby, Denmark [ah@imm.dtu.dk](mailto:ah@imm.dtu.dk)

<sup>3</sup> LFCS, School of Informatics, University of Edinburgh, Edinburgh, UK [dts@inf.ed.ac.uk](mailto:dts@inf.ed.ac.uk)

<sup>4</sup> Institute of Informatics, Warsaw University and Institute of Computer Science, Polish Academy of Science, Warsaw, Poland [tarlecki@mimuw.edu.pl](mailto:tarlecki@mimuw.edu.pl)

**Summary.** CASL is an expressive specification language that has been designed to supersede many existing algebraic specification languages and provide a standard. CASL consists of several layers, including basic (unstructured) specifications, structured specifications and architectural specifications; the latter are used to prescribe the modular structure of implementations.

We describe a simplified version of the CASL syntax, semantics and proof calculus for each of these three layers and state the corresponding soundness and completeness theorems. The layers are orthogonal in the sense that the semantics of a given layer uses that of the previous layer as a “black box”, and similarly for the proof calculi. In particular, this means that CASL can easily be adapted to other logical systems.

We conclude with a detailed example specification of a warehouse, which serves to illustrate the application of both CASL and the proof calculi for the various layers.

**Key words:** Algebraic specification, formal software development, logic, calculus, institution

## 1 Introduction

*Algebraic specification* is one of the most extensively-developed approaches in the formal methods area. The most fundamental assumption underlying algebraic specification is that programs are modelled as algebraic structures that include a collection of sets of data values together with functions over those sets. This level of abstraction is commensurate with the view that the correctness of the input/output behaviour of a program takes precedence over

all its other properties. Another common element is that specifications of programs consist mainly of logical *axioms*, usually in a logical system in which equality has a prominent role, describing the properties that the functions are required to satisfy—often just by their interrelationship. This *property-oriented* approach is in contrast to so-called *model-oriented* specifications in frameworks like VDM [28] which consist of a simple realization of the required behaviour. However, the theoretical basis of algebraic specification is largely in terms of constructions on algebraic models, so it is at the same time much more model-oriented than approaches such as those based on type theory (see e.g. [52]), where the emphasis is almost entirely on syntax and formal systems of rules, and semantic models are absent or regarded as of secondary importance.

CASL [4] is an expressive specification language that has been designed by COFI, the international *Common Framework Initiative for algebraic specification and development* [48, 18], with the goal to subsume many previous algebraic specification languages and to provide a standard language for the specification and development of modular software systems.

This chapter gives an overview of the semantic concepts and proof calculi underlying CASL. Sect. 2 starts with *institutions* and *logics*, abstract formalizations of the notion of logical system. The remaining sections follow the layers of the CASL language:

1. *Basic specifications* provide the means to write specifications in a particular institution, and provide a proof calculus for reasoning within such unstructured specifications. The institution underlying CASL, together with its proof calculus, is presented in Sects. 3 (for *many-sorted basic specifications*) and 4 (the extension to *subsorting*). Section 5 explains some of the language constructs that allow one to write down theories in this institution rather concisely.
2. *Structured specifications* express how more complex specifications are built from simpler ones (Sect. 6). The semantics and proof calculus is given in a way that is parameterized over the particular institution and proof calculus for basic specifications. Hence, the institution and proof calculus for basic specifications can be changed without the need to change anything for structured specifications.
3. *Architectural specifications*, in contrast to structured specifications, prescribe the modular structure of the *implementation*, with the possibility of enforcing a separate development of composable, reusable implementation units (Sect. 7). Again, the semantics and proof calculus at this layer is formulated in terms of the semantics and proof calculus given in the previous layers.
4. Finally, *libraries of specifications* allow the (distributed) storage and retrieval of named specifications. Since this is rather straightforward, space considerations led to the omission of this layer of CASL in the present work.

For the sake of simplicity, this chapter only covers a simplified version of CASL, and mainly introduces semantic concepts, while language constructs are only briefly treated in Sect. 5. A full account of CASL, also covering libraries of specifications, is given in [50] (see also [18, 4, 37]), while a gentle introduction is provided in [49].

## 2 Institutions and Logics

First, before considering the particular concepts underlying CASL, we recall how specification frameworks in general may be formalized in terms of so-called institutions [22].

An *institution*  $I = (\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \models)$  consists of

- a category  $\mathbf{Sign}$  of *signatures*,
- a functor  $\mathbf{Sen}: \mathbf{Sign} \rightarrow \mathbf{Set}$  giving, for each signature  $\Sigma$ , a set of *sentences*  $\mathbf{Sen}(\Sigma)$ , and for each signature morphism  $\sigma: \Sigma \rightarrow \Sigma'$ , a *sentence translation map*  $\mathbf{Sen}(\sigma): \mathbf{Sen}(\Sigma) \rightarrow \mathbf{Sen}(\Sigma')$ , where  $\mathbf{Sen}(\sigma)(\varphi)$  is often written  $\sigma(\varphi)$ ,
- a functor  $\mathbf{Mod}: \mathbf{Sign}^{op} \rightarrow \mathcal{CAT}$ <sup>5</sup> giving, for each signature  $\Sigma$ , a category of *models*  $\mathbf{Mod}(\Sigma)$ , and for each signature morphism  $\sigma: \Sigma \rightarrow \Sigma'$ , a *reduct functor*  $\mathbf{Mod}(\sigma): \mathbf{Mod}(\Sigma') \rightarrow \mathbf{Mod}(\Sigma)$ , where  $\mathbf{Mod}(\sigma)(M')$  is often written  $M'|_\sigma$ , and
- a satisfaction relation  $\models_\Sigma \subseteq |\mathbf{Mod}(\Sigma)| \times \mathbf{Sen}(\Sigma)$  for each  $\Sigma \in \mathbf{Sign}$ ,

such that for each  $\sigma: \Sigma \rightarrow \Sigma'$  in  $\mathbf{Sign}$  the following *satisfaction condition* holds:

$$M' \models_{\Sigma'} \sigma(\varphi) \iff M'|_\sigma \models_\Sigma \varphi$$

for each  $M' \in \mathbf{Mod}(\Sigma')$  and  $\varphi \in \mathbf{Sen}(\Sigma)$ .

An *institution with unions* is an institution equipped with a partial binary operation  $\cup$  on signatures, such that there are two “inclusions”  $\iota_1: \Sigma_1 \rightarrow \Sigma_1 \cup \Sigma_2$  and  $\iota_2: \Sigma_2 \rightarrow \Sigma_1 \cup \Sigma_2$ . We write  $M|_{\Sigma_i}$  for  $M|_{\iota_i: \Sigma_i \rightarrow \Sigma_1 \cup \Sigma_2}$  ( $i = 1, 2$ ) whenever  $\iota_i$  is clear from the context. Typically (e.g. in the CASL institution),  $\cup$  is a total operation. However, in institutions without overloading, generally two signatures giving the same name to different things cannot be united.

When  $\Sigma_1 \cup \Sigma_2 = \Sigma_2$  with  $\iota_2: \Sigma_2 \rightarrow (\Sigma_1 \cup \Sigma_2 = \Sigma_2)$  being identity, we say that  $\Sigma_1$  is a *subsignature* of  $\Sigma_2$ , written  $\Sigma_1 \subseteq \Sigma_2$ .

Further properties of signature unions, as well as other requirements on institutions, are needed only in Sect. 7 on architectural specifications and will be introduced there.

<sup>5</sup> Here,  $\mathcal{CAT}$  is the quasi-category of all categories. As metatheory, we use  $ZFCU$ , i.e.  $ZF$  with axiom of choice and a set-theoretic universe  $U$ . This allows for the construction of quasi-categories, i.e. categories with more than a class of objects. See [25].

Within an arbitrary but fixed institution, we can easily define the usual notion of *logical consequence* or *semantical entailment*. Given a set of  $\Sigma$ -sentences  $\Gamma$  and a  $\Sigma$ -sentence  $\varphi$ , we say that  $\varphi$  *follows from*  $\Gamma$ , written  $\Gamma \models_{\Sigma} \varphi$ , iff for all  $\Sigma$ -models  $M$ , we have  $M \models_{\Sigma} \Gamma$  implies  $M \models_{\Sigma} \varphi$ . (Here,  $M \models_{\Sigma} \Gamma$  means that  $M \models_{\Sigma} \psi$  for each  $\psi \in \Gamma$ .)

Coming to proofs, a logic [33] extends an institution with proof-theoretic entailment relations that are compatible with semantic entailment.

A logic  $\mathcal{LOG} = (\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \models, \vdash)$  is an institution  $(\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \models)$  equipped with an *entailment system*  $\vdash$ , that is, a relation  $\vdash_{\Sigma} \subseteq \mathcal{P}(\mathbf{Sen}(\Sigma)) \times \mathbf{Sen}(\Sigma)$  for each  $\Sigma \in |\mathbf{Sign}|$ , such that the following properties are satisfied for any  $\varphi \in \mathbf{Sen}(\Sigma)$  and  $\Gamma, \Gamma' \subseteq \mathbf{Sen}(\Sigma)$ :

1. *reflexivity*:  $\{\varphi\} \vdash_{\Sigma} \varphi$ ,
2. *monotonicity*: if  $\Gamma \vdash_{\Sigma} \varphi$  and  $\Gamma' \supseteq \Gamma$  then  $\Gamma' \vdash_{\Sigma} \varphi$ ,
3. *transitivity*: if  $\Gamma \vdash_{\Sigma} \varphi_i$  for  $i \in I$  and  $\Gamma \cup \{\varphi_i \mid i \in I\} \vdash_{\Sigma} \psi$ , then  $\Gamma \vdash_{\Sigma} \psi$ ,
4.  *$\vdash$ -translation*: if  $\Gamma \vdash_{\Sigma} \varphi$ , then for any  $\sigma: \Sigma \rightarrow \Sigma'$  in  $\mathbf{Sign}$ ,  $\sigma(\Gamma) \vdash_{\Sigma'} \sigma(\varphi)$ ,
5. *soundness*: if  $\Gamma \vdash_{\Sigma} \varphi$  then  $\Gamma \models_{\Sigma} \varphi$ .

A logic is *complete* if, in addition,  $\Gamma \models_{\Sigma} \varphi$  implies  $\Gamma \vdash_{\Sigma} \varphi$ .

It is easy to obtain a complete logic from an institution by simply defining  $\vdash$  as  $\models$ . Hence,  $\vdash$  might appear to be redundant. However, the point is that  $\vdash$  will typically be defined via a system of finitary *derivation rules*. This gives rise to a notion of *proof* that is absent when the institution is considered on its own, even if the relation that results coincides with semantic entailment which is defined in terms of the satisfaction relation.

### 3 Many-Sorted Basic Specifications

CASL's basic specification layer is an expressive language that integrates subsorts, partiality, first-order logic and induction (the latter expressed using so-called sort generation constraints).

#### 3.1 Many-Sorted Institution

The institution underlying CASL is introduced in two steps [9, 16]. In this section, we introduce the institution of many-sorted partial first-order logic with sort generation constraints and equality,  $PCFOL^{\bar{=}}$ . In Sect. 4, subsorting is added.

#### Signatures

A *many-sorted signature*  $\Sigma = (S, TF, PF, P)$  consists of a set  $S$  of *sorts*,  $S^* \times S$ -indexed families  $TF$  and  $PF$  of *total* and *partial function symbols*, with  $TF_{w,s} \cap PF_{w,s} = \emptyset$  for each  $(w, s) \in S^* \times S$ , where constants are treated

as functions with no arguments, and an  $S^*$ -indexed family  $P$  of *predicate symbols*. We write  $f : w \rightarrow s \in TF$  for  $f \in TF_{w,s}$  (with  $f : s$  for empty  $w$ ),  $f : w \rightarrow? s \in PF$  for  $f \in PF_{w,s}$  (with  $f : \rightarrow? s$  for empty  $w$ ) and  $p : w \in P$  for  $p \in P_w$ .

Although  $TF_{w,s}$  and  $PF_{w,s}$  are required to be disjoint, so that a function symbol with a given profile cannot be both partial and total, function and predicate symbols may be overloaded: we do not require e.g.  $TF_{w,s}$  and  $TF_{w',s'}$  (or  $TF_{w,s}$  and  $PF_{w',s'}$ ) to be disjoint for  $(w, s) \neq (w', s')$ . To ensure that there is no ambiguity in sentences, however, symbols are always qualified by profiles when used. In the CASL language constructs (see Sect. 5), such qualifications may be omitted when they are unambiguously determined by the context.

Given signatures  $\Sigma$  and  $\Sigma'$ , a *signature morphism*  $\sigma : \Sigma \rightarrow \Sigma'$  maps sorts, function symbols and predicate symbols in  $\Sigma$  to symbols of the same kind in  $\Sigma'$ . A partial function symbol may be mapped to a total function symbol, but not vice versa, and profiles must be preserved, so for instance  $f : w \rightarrow s$  in  $\Sigma$  maps to a function symbol in  $\Sigma'$  with profile  $\sigma^*(w) \rightarrow \sigma(s)$ , where  $\sigma^*$  is the extension of  $\sigma$  to finite strings of symbols. Identities and composition are defined in the obvious way, giving a category **Sign** of  $PCFOL^=$ -signatures.

## Models

Given a finite string  $w = s_1 \dots s_n$  and sets  $M_{s_1}, \dots, M_{s_n}$ , we write  $M_w$  for the Cartesian product  $M_{s_1} \times \dots \times M_{s_n}$ . Let  $\Sigma = (S, TF, PF, P)$ .

A *many-sorted  $\Sigma$ -model*  $M$  consists of a non-empty *carrier set*  $M_s$  for each sort  $s \in S$ , a total function  $(f_{w,s})_M : M_w \rightarrow M_s$  for each total function symbol  $f : w \rightarrow s \in TF$ , a partial function  $(f_{w,s})_M : M_w \rightarrow M_s$  for each partial function symbol  $f : w \rightarrow? s \in PF$ , and a predicate  $(p_w)_M \subseteq M_w$  for each predicate symbol  $p : w \in P$ . Requiring carriers to be non-empty simplifies deduction and makes it unproblematic to regard axioms (see Sect. 3.1) as implicitly universally quantified. A slight drawback is that the existence of initial models is lost in some cases, even if only equational axioms are used, namely if the signature is such that there are no ground terms of some sort. However, from a methodological point of view, specifications with such signatures are typically used in a context where loose rather than initial semantics is appropriate.

A *many-sorted  $\Sigma$ -homomorphism*  $h : M \rightarrow N$  maps the values in the carriers of  $M$  to values in the corresponding carriers of  $N$  in such a way that the values of functions and their definedness is preserved, as well as the truth of predicates. Identities and composition are defined in the obvious way. This gives a category **Mod**( $\Sigma$ ).

Concerning *reducts*, if  $\sigma : \Sigma \rightarrow \Sigma'$  is a signature morphism and  $M'$  is a  $\Sigma'$ -model, then  $M'|_\sigma$  is a  $\Sigma$ -model with  $(M'|_\sigma)_s := M'_{\sigma(s)}$  for  $s \in S$  and analogously for  $(f_{w,s})_{M'|_\sigma}$  and  $(p_w)_{M'|_\sigma}$ . The same applies to any  $\Sigma'$ -homomorphism  $h' : M' \rightarrow N'$ : its reduct  $h'|_\sigma : M'|_\sigma \rightarrow N'|_\sigma$  is the  $\Sigma$ -

homomorphism defined by  $(h'|_\sigma)_s := h'_{\sigma(s)}$  for  $s \in S$ . It is easy to see that reduct preserves identities and composition, so we obtain a functor  $\mathbf{Mod}(\sigma): \mathbf{Mod}(\Sigma') \rightarrow \mathbf{Mod}(\Sigma)$ . Moreover, it is easy to see that reducts are compositional, i.e., we have, for example,  $(M''|_\theta)|_\sigma = M''|_{\sigma;\theta}$  for all signature morphisms  $\sigma: \Sigma \rightarrow \Sigma'$ ,  $\theta: \Sigma' \rightarrow \Sigma''$  and  $\Sigma''$ -models  $M''$ . This means that we have indeed defined a functor  $\mathbf{Mod}: \mathbf{Sign}^{op} \rightarrow \mathcal{CAT}$ .

### Sentences

Let  $\Sigma = (S, TF, PF, P)$ . A *variable system* over  $\Sigma$  is an  $S$ -sorted, pairwise disjoint family of variables  $X = (X_s)_{s \in S}$ . Let such a variable system be given.

As usual, the *many-sorted  $\Sigma$ -terms* over  $X$  are defined inductively as comprising the variables in  $X$ , which have uniquely-determined sorts, together with applications of function symbols to argument terms of appropriate sorts, where the sort is determined by the profile of its outermost function symbol. This gives an  $S$ -indexed family of sets  $T_\Sigma(X)$  which can be made into a (total) many-sorted  $\Sigma$ -model by defining  $(f_{w,s})_{T_\Sigma(X)}$  to be the term-formation operations for  $f: w \rightarrow s \in TF$  and  $f: w \rightarrow ? \ s \in PF$ , and  $(p_w)_{T_\Sigma(X)} = \emptyset$  for  $p: w \in P$ .

An atomic  $\Sigma$ -formula is either: an application  $p_w(t_1, \dots, t_n)$  of a predicate symbol to terms of appropriate sorts; an *existential equation*  $t \stackrel{e}{=} t'$  or *strong equation*  $t \stackrel{s}{=} t'$  between two terms of the same sort; or an assertion *def*  $t$  that the value of a term is defined. This defines the set  $AF_\Sigma(X)$  of *many-sorted atomic  $\Sigma$ -formulas* with variables in  $X$ . The set  $FO_\Sigma(X)$  of *many-sorted first-order  $\Sigma$ -formulas* with variables in  $X$  is then defined by adding a formula *false* and closing under implication  $\varphi \Rightarrow \psi$  and universal quantification  $\forall x: s \bullet \varphi$ . We use the usual abbreviations:  $\neg\varphi$  for  $\varphi \Rightarrow \text{false}$ ,  $\varphi \wedge \psi$  for  $\neg(\varphi \Rightarrow \neg\psi)$ ,  $\varphi \vee \psi$  for  $\neg(\neg\varphi \wedge \neg\psi)$ , *true* for  $\neg\text{false}$  and  $\exists x: s \bullet \varphi$  for  $\neg\forall x: s \bullet \neg\varphi$ .

A *sort generation constraint* states that a given set of sorts is generated by a given set of functions. Technically, sort generation constraints also contain a signature morphism component; this allows them to be translated along signature morphisms without sacrificing the satisfaction condition. Formally, a sort generation constraint over a signature  $\Sigma$  is a triple  $(\tilde{S}, \tilde{F}, \theta)$ , where  $\theta: \tilde{\Sigma} \rightarrow \Sigma$ ,  $\tilde{\Sigma} = (\tilde{S}, \tilde{TF}, \tilde{PF}, \tilde{P})$ ,  $\tilde{S} \subseteq \Sigma$  and  $\tilde{F} \subseteq TF \cup PF$ .

Now a  $\Sigma$ -sentence is either a closed many-sorted first-order  $\Sigma$ -formula (i.e. a many-sorted first-order  $\Sigma$ -formula over the empty set of variables), or a sort generation constraint over  $\Sigma$ .

Given a signature morphism  $\sigma: \Sigma \rightarrow \Sigma'$  and variable system  $X$  over  $\Sigma$ , we can get a variable system  $\sigma(X)$  over  $\Sigma'$  by taking

$$\sigma(X)_{s'} := \bigcup_{\sigma(s)=s'} X_s$$

Since  $T_\Sigma(X)$  is total, the inclusion  $\zeta_{\sigma,X}: X \rightarrow T_{\Sigma'}(\sigma(X))|_\sigma$  (regarded as a variable valuation) leads to a term evaluation function

$$\zeta_{\sigma, X}^{\#} : T_{\Sigma}(X) \rightarrow T_{\Sigma'}(\sigma(X))|_{\sigma}$$

that is total as well. This can be inductively extended to a translation along  $\sigma$  of  $\Sigma$ -first order formulas with variables in  $X$  by taking  $\sigma(t) := \zeta_{\sigma, X}^{\#}(t)$ ,  $\sigma(p_w(t_1, \dots, t_n)) := \sigma_w(p)_{\sigma^*(w)}(\sigma(t_1), \dots, \sigma(t_n))$ ,  $\sigma(t \stackrel{e}{=} t') := \sigma(t) \stackrel{e}{=} \sigma(t')$ ,  $\sigma(\forall x : s \bullet \varphi) = \forall x : \sigma(s) \bullet \sigma(\varphi)$ , and so on. The translation of a  $\Sigma$ -constraint  $(\tilde{S}, \tilde{F}, \theta)$  along  $\sigma$  is the  $\Sigma'$ -constraint  $(\tilde{S}, \tilde{F}, \theta; \sigma)$ . It is easy to see that sentence translation preserves identities and composition, so sentence translation is functorial.

### Satisfaction

Variable valuations are total, but the value of a term with respect to a variable valuation may be undefined, due to the application of a partial function during the evaluation of the term. Given a variable valuation  $\nu : X \rightarrow M$  for  $X$  in  $M$ , *term evaluation*  $\nu^{\#} : T_{\Sigma}(X) \rightarrow M$  is defined in the obvious way, with  $t \in \text{dom}(\nu^{\#})$  iff all partial functions in  $t$  are applied to values in their domains.

Even though the evaluation of a term with respect to a variable valuation may be undefined, the satisfaction of a formula  $\varphi$  in a model  $M$  is always defined, and it is either true or false: that is, we have a two-valued logic. The application  $p_w(t_1, \dots, t_n)$  of a predicate symbol to a sequence of argument terms is satisfied with respect to a valuation  $\nu : X \rightarrow M$  iff the values of all of  $t_1, \dots, t_n$  are defined under  $\nu^{\#}$  and give a tuple belonging to  $p_M$ . A definedness assertion *def*  $t$  is satisfied iff the value of  $t$  is defined. An existential equation  $t_1 \stackrel{e}{=} t_2$  is satisfied iff the values of  $t_1$  and  $t_2$  are defined and equal, whereas a strong equation  $t_1 \stackrel{s}{=} t_2$  is also satisfied when the values of both  $t_1$  and  $t_2$  are undefined; thus both kinds of equation coincide for defined terms. Satisfaction of other formulae is defined in the obvious way. A formula  $\varphi$  is satisfied in a model  $M$ , written  $M \models \varphi$ , iff it is satisfied with respect to all variable valuations into  $M$ .

A  $\Sigma$ -constraint  $(\tilde{S}, \tilde{F}, \theta)$  is satisfied in a  $\Sigma$ -model  $M$  iff the carriers of  $M|_{\theta}$  of sorts in  $\tilde{S}$  are generated by the function symbols in  $\tilde{F}$ , i.e. for every sort  $s \in \tilde{S}$  and every value  $a \in (M|_{\theta})_s$ , there is a  $\tilde{\Sigma}$ -term  $t$  containing only function symbols from  $\tilde{F}$  and variables of sorts not in  $\tilde{S}$  such that  $\nu^{\#}(t) = a$  for some valuation  $\nu$  into  $M|_{\theta}$ .

For a sort generation constraint  $(\tilde{S}, \tilde{F}, \theta)$  we can assume without loss of generality that all the result sorts of function symbols in  $\tilde{F}$  occur in  $\tilde{S}$ . If not, we can just omit from  $\tilde{F}$  those function symbols not satisfying this requirement, without affecting satisfaction of the sort generation constraint: in the  $\tilde{\Sigma}$ -term  $t$  witnessing the satisfaction of the constraint, any application of a function symbol with result sort outside  $\tilde{S}$  can be replaced by a variable of that sort, which gets as assigned value the evaluation of the function application.

For a proof of the satisfaction condition, see [37].

### 3.2 Proof Calculus

We now come to the proof calculus for CASL many-sorted basic specification. The rules of derivation are given in Figs. 1 and 2.

<p>(Absurdity) <math>\frac{false}{\varphi}</math></p>	<p>(Tertium non datur) <math>\frac{\begin{array}{c} [\varphi] \quad [\varphi \Rightarrow false] \\ \vdots \quad \vdots \\ \psi \quad \psi \end{array}}{\psi}</math></p>	
<p>(<math>\Rightarrow</math>-intro) <math>\frac{\begin{array}{c} [\varphi] \\ \vdots \\ \psi \end{array}}{\varphi \Rightarrow \psi}</math></p>	<p>(<math>\Rightarrow</math>-elim) <math>\frac{\varphi \quad \varphi \Rightarrow \psi}{\psi}</math></p>	<p>(<math>\forall</math>-elim) <math>\frac{\forall x : s. \varphi}{\varphi}</math></p>
<p>(<math>\forall</math>-intro) <math>\frac{\varphi}{\forall x : s. \varphi}</math> where <math>x_s</math> occurs freely only in local assumptions</p>		
<p>(Reflexivity) <math>\frac{}{x_s \stackrel{e}{=} x_s}</math> if <math>x_s</math> is a variable</p>		
<p>(Congruence) <math>\frac{\varphi}{(\bigwedge_{x_s \in FV(\varphi)} x_s \stackrel{e}{=} \nu(x_s)) \Rightarrow \varphi[\nu]}</math> if <math>\varphi[\nu]</math> defined</p>		
<p>(Substitution) <math>\frac{\varphi}{(\bigwedge_{x_s \in FV(\varphi)} D(\nu(x_s))) \Rightarrow \varphi[\nu]}</math> if <math>\varphi[\nu]</math> defined and <math>FV(\varphi)</math> occur freely only in local assumptions</p>		
<p>(Totality) <math>\frac{}{D(f_{w,s}(x_{s_1}, \dots, x_{s_n}))}</math> if <math>w = s_1 \dots s_n, f \in TF_{w,s}</math></p>		
<p>(Function Strictness) <math>\frac{t_1 \stackrel{e}{=} t_2}{D(t)}</math> <math>t</math> some subterm of <math>t_1</math> or <math>t_2</math></p>		
<p>(Predicate Strictness) <math>\frac{p_w(t_1, \dots, t_n)}{D(t_i)}</math> <math>i \in \{1, \dots, n\}</math></p>		

**Fig. 1.** First-order deduction rules for CASL basic specifications

The first rules (up to  $\forall$ -intro) are standard rules of first-order logic [6]. Reflexivity, Congruence and Substitution differ from the standard rules since they have to take into account potential undefinedness of terms. Hence, Reflexivity only holds for variables (which by definition are always defined), and Substitution needs the assumption that the terms being substituted are defined. (Note that definedness,  $D(t)$ , is just an abbreviation for the existential equality  $t \stackrel{e}{=} t$ .) Totality, Function Strictness and Predicate Strictness have self-explanatory names; they allow definedness statements to be inferred. Finally, the two rules in Fig. 2 deal with sort generation constraints. If these are

$$\begin{array}{c}
(S, F, \theta: \bar{\Sigma} \rightarrow \Sigma) \\
\text{(Induction)} \quad \frac{\varphi_1 \wedge \cdots \wedge \varphi_k}{\bigwedge_{s \in S} \forall x: \theta(s) \bullet \Psi_s(x)} \\
F = \{f_1: s_1^1 \dots s_{m_1}^1 \rightarrow s^1; \dots; f_k: s_1^k \dots s_{m_k}^k \rightarrow s^k\}, \\
\Psi_{s_j} \text{ is a formula with one free variable } x \text{ of sort } \theta(s_j), j = 1, \dots, k, \\
\varphi_j = \forall x_1: \theta(s_1^j), \dots, x_{m_j}: \theta(s_{m_j}^j) \bullet \\
\quad \left( D(\theta(f_j)(x_1, \dots, x_{m_j})) \wedge \bigwedge_{i=1, \dots, m_j; s_i^j \in S} \Psi_{s_i^j}(x_i) \right) \\
\quad \Rightarrow \Psi_{s_j}(\theta(f_j)(x_1, \dots, x_{m_j})) \\
\\
\text{(Sortgen-intro)} \quad \frac{\varphi_1 \wedge \cdots \wedge \varphi_k \Rightarrow \bigwedge_{s \in S} \forall x: \theta(s) \bullet p_s(x)}{(S, F, \theta: \bar{\Sigma} \rightarrow \Sigma)} \\
F = \{f_1: s_1^1 \dots s_{m_1}^1 \rightarrow s^1; \dots; f_k: s_1^k \dots s_{m_k}^k \rightarrow s^k\}, \\
\text{for } s \in S, \text{ the predicates } p_s: \theta(s) \text{ occur only in local assumptions,} \\
\text{and for } j = 1, \dots, k, \\
\varphi_j = \forall x_1: \theta(s_1^j), \dots, x_{m_j}: \theta(s_{m_j}^j) \bullet \\
\quad \left( D(\theta(f_j)(x_1, \dots, x_{m_j})) \wedge \bigwedge_{i=1, \dots, m_j; s_i^j \in S} p_{s_i^j}(x_i) \right) \\
\quad \Rightarrow p_{s_j}(\theta(f_j)(x_1, \dots, x_{m_j}))
\end{array}$$

Fig. 2. Induction rules for CASL basic specifications

seen as second-order universally quantified formulas, Induction corresponds to second-order  $\forall$ -Elim, and Sortgen-Intro corresponds to second-order  $\forall$ -Intro. The  $\varphi_j$  correspond to the inductive bases and inductive steps that have to be shown, while the formula  $\bigwedge_{s \in S} \forall x: \theta(s) \bullet \Psi_s(x)$  is the statement that is shown by induction. Note that if  $S$  consists of more than one sort, we have a simultaneous induction over several sorts.

A *derivation* of  $\Phi \vdash \varphi$  is a tree (called a *derivation tree*) such that

- the root of the tree is  $\varphi$ ,
- all the leaves of the tree are either in  $\Phi$  or marked as local assumptions,
- each non-leaf node is an instance of the conclusion of some rule, with its children being the correspondingly instantiated premises, and
- any assumptions marked with [...] in the proof rules are marked as local assumptions.

If  $\Phi$  and  $\varphi$  consist of  $\Sigma$ -formulas, we also write  $\Phi \vdash_{\Sigma} \varphi$ . In practice, one will work with acyclic graphs instead of trees, since this allows the re-use of lemmas.

Some rules contain a condition that some variables occur freely only in local assumptions. These conditions are the usual eigenvariable conditions of natural deduction style calculi. They more precisely mean that if the specified variables occur freely in an assumption in a proof tree, the assumption must be marked as local and have been used in the proof of the premise of the respective rule.

In order to carry out a proof in the calculus, it is convenient to prove some derived rules. We only list a few here:

$$\begin{aligned} (\wedge\text{-Intro}) \quad & \frac{\varphi \quad \psi}{\varphi \wedge \psi} \\ (\wedge\text{-Elim1}) \quad & \frac{\varphi \wedge \psi}{\varphi} \\ (\wedge\text{-Elim2}) \quad & \frac{\varphi \wedge \psi}{\psi} \end{aligned}$$

Recall that  $\varphi \wedge \psi$  is defined<sup>6</sup> to be  $(\varphi \Rightarrow \psi \Rightarrow \text{false}) \Rightarrow \text{false}$ .<sup>7</sup>

Proof of **( $\wedge$ -Intro)**: Assume  $\varphi$  and  $\psi$ . Further assume (aiming at a proof using **( $\Rightarrow$ -intro)**) that  $\varphi \Rightarrow \psi \Rightarrow \text{false}$ . By **( $\Rightarrow$ -elim)** twice, we get  $\text{false}$ . Hence,  $(\varphi \Rightarrow \psi \Rightarrow \text{false}) \Rightarrow \text{false}$  by **( $\Rightarrow$ -intro)**.  $\square$

Proof of **( $\wedge$ -Elim1)**: Assume  $(\varphi \Rightarrow \psi \Rightarrow \text{false}) \Rightarrow \text{false}$ . We want to prove  $\varphi$  using **(Tertium non datur)**. Obviously,  $\varphi$  can be proved from itself. It remains to prove it from  $\varphi \Rightarrow \text{false}$ . Now from  $\varphi \Rightarrow \text{false}$ , using **( $\Rightarrow$ -intro)** twice and **( $\Rightarrow$ -elim)** once, we get  $\varphi \Rightarrow \psi \Rightarrow \text{false}$ . Hence, by **( $\Rightarrow$ -elim)** with our main assumption, we get  $\text{false}$ . With **(Absurdity)**, we get  $\varphi$ . The proof of **( $\wedge$ -Elim2)** is similar.  $\square$

The following theorem is proved in [44]:

**Theorem 3.1.** *The above proof calculus yields an entailment system. Equipped with this entailment system, the CASL institution  $PCFOL^=$  becomes a sound logic. Moreover, it is complete if sort generation constraints are not used.*

With sort generation constraints, inductive datatypes such as the natural numbers can be specified monomorphically (i.e., up to isomorphism). By Gödel's incompleteness theorem, there cannot be a recursively axiomatized complete calculus for such systems.

**Theorem 3.2.** *If sort generation constraints are used, the CASL logic is not complete. Moreover, there cannot be a recursively axiomatized sound and complete entailment system for many-sorted CASL basic specifications.*

Instead of using the above calculus, it is also possible to use an encoding of the CASL logic into second order logic, see [37].

<sup>6</sup> This is not the same definition as in [50], but it allows us to keep things simple.

The proofs would also go through with the definitions of [50], albeit would be a bit more complex then.

<sup>7</sup> Note that  $\Rightarrow$  associates to the right.

## 4 Subsorted Basic Specifications

CASL allows the user to declare a sort as a subsort of another. In contrast to most other subsorted languages, CASL interprets subsorts as injective embeddings between carriers – not necessarily as inclusions. This allows for more general models in which values of a subsort are represented differently from values of the supersort, an example being integers (represented as 32-bit words) as a subsort of reals (represented using floating point). Furthermore, to avoid problems with modularity (as described in [24, 34]), there are no requirements like monotonicity, regularity or local filtration imposed on signatures. Instead, the use of overloaded functions and predicates in formulae of the CASL language is required to be sufficiently disambiguated, such that all parses have the same semantics.

### 4.1 Subsorted Institution

In order to cope with subsorting, the institution for basic specifications presented in Sect. 3 has to be modified slightly. First a category of subsorted signatures is defined (each signature is extended with a pre-order  $\leq$  on its set of sorts) and a functor from this category into the category of many-sorted signatures is defined. Then the notions of models, sentences and satisfaction can be borrowed from the many-sorted institution via this functor. Technical details follow below, leading to the institution of subsorted partial first-order logic with sort generation constraints and equality ( $SubPCFOL^=$ ).

#### Signatures

A *subsorted signature*  $\Sigma = (S, TF, PF, P, \leq)$  consists of a many-sorted signature  $(S, TF, PF, P)$  together with a reflexive transitive *subsort relation*  $\leq$  on the set  $S$  of sorts.

For a subsorted signature, we define *overloading relations* for function and predicate symbols: two function symbols  $f : w_1 \rightarrow s_1$  (or  $f : w_1 \rightarrow? s_1$ ) and  $f : w_2 \rightarrow s_2$  (or  $f : w_2 \rightarrow? s_2$ ) are in the *overloading relation* iff there exists a  $w \in S^*$  and  $s \in S$  such that  $w \leq w_1, w_2$  and  $s_1, s_2 \leq s$ . Similarly, two qualified predicate symbols  $p : w_1$  and  $p : w_2$  are in the overloading relation iff there exists a  $w \in S^*$  such that  $w \leq w_1, w_2$ .

Let  $\Sigma = (S, TF, PF, P, \leq)$  and  $\Sigma' = (S', TF', PF', P', \leq')$  be subsorted signatures. A *subsorted signature morphism*  $\sigma : \Sigma \rightarrow \Sigma'$  is a many-sorted signature morphism from  $(S, TF, PF, P)$  into  $(S', TF', PF', P')$  preserving the subsort relation and the overloading relations.

With each subsorted signature  $\Sigma = (S, TF, PF, P, \leq)$  we associate a many-sorted signature  $\widehat{\Sigma}$ , which is the extension of the underlying many-sorted signature  $(S, TF, PF, P)$  with

- a total *embedding* function symbol  $em : s \rightarrow s'$  for each pair of sorts  $s \leq s'$ ,

- a partial *projection* function symbol  $pr : s' \rightarrow? s$  for each pair of sorts  $s \leq s'$ , and
- a unary *membership* predicate symbol  $in(s) : s'$  for each pair of sorts  $s \leq s'$ .

It is assumed that the symbols used for injection, projection and membership are distinct and not used otherwise in  $\Sigma$ .

In a similar way, any subsorted signature morphism  $\sigma$  from  $\Sigma$  into  $\Sigma'$  extends to a many-sorted signature morphism  $\widehat{\sigma}$  from  $\widehat{\Sigma}$  into  $\widehat{\Sigma}'$ .

The construction  $\widehat{\phantom{x}}$  is a functor from the category of subsorted signatures **SubSig** into the category of many-sorted signatures **Sign**.

## Models

For a subsorted signature  $\Sigma = (S, TF, PF, P, \leq)$ , with embedding symbols  $em$ , projection symbols  $pr$ , and membership symbols  $in$ , the *subsorted models* for  $\Sigma$  are ordinary many-sorted models for  $\widehat{\Sigma}$  satisfying a set  $Ax(\Sigma)$  of sentences (formalised in [50], section III.3.1.2) ensuring that:

- Embedding functions are injective.
- The embedding of a sort into itself is the identity function.
- All compositions of embedding functions between the same two sorts are equal functions.
- Projection functions are injective when defined.
- Embedding followed by projection is identity.
- Membership in a subsort holds just when the projection to the subsort is defined.
- Embedding is compatible with those functions and predicates that are in the overloading relations.

*Subsorted  $\Sigma$ -homomorphisms* are ordinary many-sorted  $\widehat{\Sigma}$ -homomorphisms.

Hence, the category of subsorted  $\Sigma$ -models **SubMod**( $\Sigma$ ) is a full subcategory of **Mod**( $\widehat{\Sigma}$ ), i.e. **SubMod**( $\Sigma$ ) = **Mod**( $\widehat{\Sigma}$ ,  $Ax(\Sigma)$ ).

The *reduct* of  $\Sigma'$ -models and  $\Sigma'$ -homomorphisms along a subsorted signature morphism  $\sigma$  from  $\Sigma$  into  $\Sigma'$  is the many-sorted reduct along the signature morphism  $\widehat{\sigma}$ . Since subsorted signature morphisms preserve the overloading relations, this is well-defined and leads to a functor **Mod**( $\widehat{\sigma}$ ): **SubMod**( $\Sigma'$ )  $\rightarrow$  **SubMod**( $\Sigma$ ).

## Sentences

For a subsorted signature  $\Sigma$ , the *subsorted sentences* are the ordinary many-sorted sentences for the associated many-sorted signature  $\widehat{\Sigma}$ .

Moreover, the *subsorted translation of sentences* along a subsorted signature morphism  $\sigma$  is the ordinary many-sorted translation along  $\widehat{\sigma}$ .

The syntax of the CASL language (cf. Sect. 5) allows the user to omit subsort injections, thus permitting the axioms to be written in a simpler and more intuitive way. Static analysis then determines the corresponding sentences of the underlying institution by inserting the appropriate injections.

### Satisfaction

Since subsorted  $\Sigma$ -models and  $\Sigma$ -sentences are just certain many-sorted  $\widehat{\Sigma}$ -models and  $\widehat{\Sigma}$ -sentences, the notion of *satisfaction* for the subsorted case follows directly from the notion of satisfaction for the many-sorted case. Since reducts and sentence translation are ordinary many-sorted reducts and sentence translation, the satisfaction condition is satisfied for the subsorted case as well.

### 4.2 Borrowing of Proofs

The proof calculus can be borrowed from the many-sorted case. To prove that a  $\Sigma$ -sentence  $\varphi$  is a  $\Sigma$ -consequence of a set of assumptions  $\Phi$ , one just has to prove that  $\varphi$  is a  $\widehat{\Sigma}$ -consequence of  $\Phi$  and  $Ax(\Sigma)$ , i.e.

$$\Phi \vdash_{\Sigma} \varphi$$

if and only if

$$\Phi \cup Ax(\Sigma) \vdash_{\widehat{\Sigma}} \varphi$$

Soundness and (for the sublogic without sort generation constraints) completeness follows from the many-sorted case.

## 5 CASL Language Constructs

Since the level of syntactic constructs will be treated only informally in this chapter, we just give a brief overview of the constructs for writing basic specifications (i.e. specifications in-the-small) in CASL. A detailed description can be found in the CASL Language Summary [30] and the CASL semantics [9].

The CASL language provides constructs for declaring sorts, subsorts, operations<sup>8</sup> and predicates that contribute to the signature in the obvious way. Operations, predicates and subsorts can also be defined in terms of others; this leads to a corresponding declaration plus a defining axiom.

Operation and predicate symbols may be overloaded; this can lead to ambiguities in formulas. A formula is well-formed only if there is a unique way of consistently adding profile qualifications, up to equivalence with respect to the overloading relations.

<sup>8</sup> At the level of syntactic constructs, functions are called operations.

```

%list [_], nil, _ :: _
%prec { _ :: _ } < { _ ++ _ }

spec LIST [sort Elem] =
  free type List[Elem] ::= nil | _ :: _ (head :? Elem; tail :? List[Elem]);
  sort NEList[Elem] = { L : List[Elem] • ¬L = nil };
  op ++ : List[Elem] × List[Elem] → List[Elem];
  forall e : Elem; K, L : List[Elem]
    • nil ++ L = L                                     %(concat_nil)%
    • (e :: K) ++ L = e :: K ++ L                       %(concat_cons)%
end

```

Fig. 3. Specification of lists over an arbitrary element sort in CASL

Binary operations can be declared to be associative, commutative, idempotent, or to have a unit. This leads to a corresponding axiom, and, in the case of associativity, to an associativity annotation.

For operations and predicates, mixfix syntax is provided. Precedence and associativity annotations may help to disambiguate terms containing mixfix symbols. There is also a syntax for literals such as numbers and strings, which allows the usual datatypes to be specified purely in CASL, without the need for magic built-in modules.

The **type**, **free type** and **generated type** constructs allow the concise description of datatypes. These are expanded into the declaration of the corresponding constructor and selector operations and axioms relating the selectors and constructors. In the case of generated and free datatypes, a sort generation constraint is also produced. Free datatypes additionally lead to axioms that assert the injectivity of the constructors and the disjointness of their images.

A typical CASL specification is shown in Fig. 3. The translation of CASL constructs to the underlying mathematical concepts is formally defined in the CASL semantics [9], which gives the semantics of language constructs in two parts. The *static semantics* checks well-formedness of a specification and produces a signature as result, failing to produce any result for ill-formed phrases. The *model semantics* provides the corresponding model-theoretic part of the semantics and produces a class of models as a result, and is intended to be applied only to phrases that are well-formed according to the static semantics. A statically well-formed phrase may still be ill-formed according to the model semantics, and then no result is produced.

## 6 Structured Specifications

The CASL structuring concepts and constructs and their semantics do not depend on a specific framework of basic specifications. This means that the

design of many-sorted and subsorted CASL specifications as explained in the previous sections is orthogonal to the design of structured specifications that we are now going to describe (this also holds for the remaining parts of CASL: architectural specifications and libraries). In this way, we achieve that the CASL basic specifications as given above can be restricted to sublanguages or extended in various ways (or even replaced completely) without the need to reconsider or to change syntax and semantics of structured specifications. The central idea for achieving this form of genericity is the notion of institution introduced in Sect. 2. Indeed, many different logics, including first-order [22], higher-order [14], polymorphic [51], modal [17, 66], temporal [21], process [21], behavioural [11, 54], coalgebraic [47] and object-oriented [64, 23, 31, 65, 2] logics have been shown to be institutions.

<pre> SPEC ::= BASIC-SPEC         SPEC<sub>1</sub> and SPEC<sub>2</sub>         SPEC with <math>\sigma</math>         SPEC hide <math>\sigma</math>         SPEC<sub>1</sub> then free { SPEC<sub>2</sub> } </pre>
--

Fig. 4. Simplified syntax of CASL structured specifications

## 6.1 Syntax and Semantics of Structured Specifications

Given an arbitrary but fixed institution with unions, it is now possible to define *structured specifications*. Their syntax is given in Fig. 4. The syntax of basic specifications **BASIC-SPEC** (as well as that of signature morphisms  $\sigma$ ) is left unexplained, since it is provided together with the institution.

Fig. 5 shows the semantics of structured specifications [60, 9]. The static semantics is shown on the left of the figure, using judgements of the form  $\vdash \textit{phrase} \triangleright \textit{result}$  (read: *phrase* statically elaborates to *result*). The model semantics is shown on the right, using judgements of the form  $\vdash \textit{phrase} \Rightarrow \textit{result}$  (read: *phrase* evaluates to *result*).

As expected, we assume that every basic specification (statically) determines a signatures and a (finite) set of axioms, which in turn determine the class of models of this specification.

Using the model semantics, we can define semantical entailment as follows: a well-formed  $\Sigma$ -specification  $SP$  entails a  $\Sigma$ -sentence  $\varphi$ , written  $SP \models_{\Sigma} \varphi$ , if  $\varphi$  is satisfied in all  $SP$ -models. A specification is consistent if its model class is non-empty. Moreover, we also have a simple notion of refinement between specifications:  $SP_1$  refines to  $SP_2$ , written  $SP_1 \approx SP_2$ , if every  $SP_2$ -model is also an  $SP_1$ -model. Given a  $\Sigma_1$ -specification  $SP_1$  and a  $\Sigma_2$ -specification  $SP_2$ , a *specification morphism*  $\sigma: SP_1 \rightarrow SP_2$  is a signature morphism  $\sigma: \Sigma_1 \rightarrow \Sigma_2$

$\frac{\vdash \mathbf{BASIC-SPEC} \triangleright \langle \Sigma, \Gamma \rangle}{\vdash \mathbf{BASIC-SPEC} \triangleright \Sigma}$	$\frac{\begin{array}{c} \vdash \mathbf{BASIC-SPEC} \triangleright \langle \Sigma, \Gamma \rangle \\ \mathcal{M} = \{M \in \mathbf{Mod}(\Sigma) \mid M \models \Gamma\} \end{array}}{\vdash \mathbf{BASIC-SPEC} \Rightarrow \mathcal{M}}$
$\frac{\begin{array}{c} \vdash SP_1 \triangleright \Sigma_1 \\ \vdash SP_2 \triangleright \Sigma_2 \\ \Sigma_1 \cup \Sigma_2 \text{ is defined} \end{array}}{\vdash SP_1 \text{ and } SP_2 \triangleright \Sigma_1 \cup \Sigma_2}$	$\frac{\begin{array}{c} \vdash SP_1 \triangleright \Sigma_1 \quad \vdash SP_2 \triangleright \Sigma_2 \\ \Sigma' = \Sigma_1 \cup \Sigma_2 \text{ is defined} \\ \vdash SP_1 \Rightarrow \mathcal{M}_1 \quad \vdash SP_2 \Rightarrow \mathcal{M}_2 \\ \mathcal{M} = \{M \in \mathbf{Mod}(\Sigma') \mid M _{\Sigma_i} \in \mathcal{M}_i, i = 1, 2\} \end{array}}{\vdash SP_1 \text{ and } SP_2 \Rightarrow \mathcal{M}}$
$\frac{\vdash SP \triangleright \Sigma}{\vdash SP \text{ with } \sigma: \Sigma \rightarrow \Sigma' \triangleright \Sigma'}$	$\frac{\begin{array}{c} \vdash SP \triangleright \Sigma \quad \vdash SP \Rightarrow \mathcal{M} \\ \mathcal{M}' = \{M \in \mathbf{Mod}(\Sigma') \mid M _{\sigma} \in \mathcal{M}\} \end{array}}{\vdash SP \text{ with } \sigma: \Sigma \rightarrow \Sigma' \Rightarrow \mathcal{M}'}$
$\frac{\vdash SP \triangleright \Sigma'}{\vdash SP \text{ hide } \sigma: \Sigma \rightarrow \Sigma' \triangleright \Sigma}$	$\frac{\begin{array}{c} \vdash SP \triangleright \Sigma' \quad \vdash SP \Rightarrow \mathcal{M} \\ \mathcal{M}' = \{M _{\sigma} \mid M \in \mathcal{M}\} \end{array}}{\vdash SP \text{ hide } \sigma: \Sigma \rightarrow \Sigma' \Rightarrow \mathcal{M}'}$
$\frac{\begin{array}{c} \vdash SP_1 \triangleright \Sigma_1 \quad \vdash SP_2 \triangleright \Sigma_2 \\ \Sigma_1 \subseteq \Sigma_2 \end{array}}{\vdash SP_1 \text{ then free } \{SP_2\} \triangleright \Sigma_2}$	$\frac{\begin{array}{c} \vdash SP_1 \triangleright \Sigma_1 \quad \vdash SP_2 \triangleright \Sigma_2 \\ \iota: \Sigma_1 \rightarrow \Sigma_2 \text{ is the inclusion} \\ \vdash SP_1 \Rightarrow \mathcal{M}_1 \quad \vdash SP_2 \Rightarrow \mathcal{M}_2 \\ \mathcal{M}' = \{M \mid M \text{ is } \mathbf{Mod}(\iota)\text{-free over } M _{\iota} \text{ in } \mathcal{M}_2\} \end{array}}{\vdash SP_1 \text{ then free } \{SP_2\} \Rightarrow \mathcal{M}'}$
<p><math>M</math> being <math>\mathbf{Mod}(\iota)</math>-free over <math>M _{\iota}</math> in <math>\mathcal{M}_2</math> means that for each model <math>M' \in \mathcal{M}_2</math> and model morphism <math>h: M _{\iota} \rightarrow M' _{\iota}</math>, there exists a unique model morphism <math>h^\#: M \rightarrow M'</math> with <math>h^\# _{\iota} = h</math>.</p>	

Fig. 5. Semantics of structured specifications

such that for each  $SP_2$ -model  $M$ ,  $M|_{\sigma}$  is an  $SP_1$ -model. Note that  $\sigma: SP_1 \rightarrow SP_2$  is a specification morphism iff  $SP_1 \approx SP_2 \text{ hide } \sigma$ .

The above language is a somewhat simplified version of CASL structured specifications. The first simplification concerns the way signature morphisms are given. It is quite inconvenient to be forced to always write down a complete signature morphism, listing explicitly how each fully qualified symbol is mapped. As a solution to this problem, CASL provides a notion of *symbol maps*, based on an appropriate notion of *institution with symbols*. Symbol maps are a very concise notation for signature morphisms. Qualifications with profiles, symbols that are mapped identically and even those whose mapping is determined uniquely may be omitted.

The second simplification concerns the fact that it is often very convenient to define specifications as extensions of existing specifications. For example, in **SPEC then free { SPEC' }**, typically SPEC' is an extension of SPEC, and one does not really want to repeat all the declarations in SPEC again in SPEC' just for the sake of turning SPEC' into a self-contained specification. Therefore, CASL has a construct **SP then SP'**, where SP' can be a *specification fragment* that is interpreted in the context (referred to as the *local environment*) coming from SP. This extension construct can be simulated using a translation along a signature inclusion and a union.

Details concerning these features can be found in [49, 9, 35].

## 6.2 A Proof Calculus for Structured Specifications

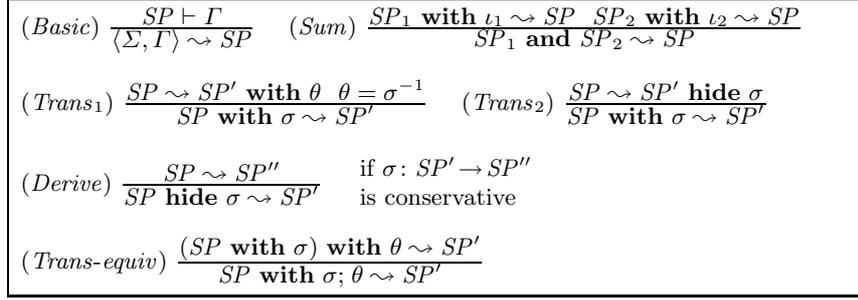
As explained above, the semantics of CASL structured specifications is parameterized over an institution providing the semantics of basic specifications. The situation with the proof calculus is similar: here, we need a logic, i.e. an institution equipped with an entailment system. Based on this, it is possible to design a logic independent proof calculus [15] for proving entailments of the form  $SP \vdash \varphi$ , where SP is a structured specification and  $\varphi$  is a formula, see Fig. 6. Figure 7 shows an extension of the structured proof calculus to refinements between specifications. Note that for the latter calculus, an *oracle for conservative extensions* is needed. A specification morphism  $\sigma: SP_1 \rightarrow SP_2$  is conservative iff each  $SP_1$ -model is the  $\sigma$ -reduct of some  $SP_2$ -model.<sup>9</sup>

$(CR) \frac{\{SP \vdash \varphi_i\}_{i \in I} \quad \{\varphi_i\}_{i \in I} \vdash \varphi}{SP \vdash \varphi}$	$(basic) \frac{\varphi \in \Gamma}{\langle \Sigma, \Gamma \rangle \vdash \varphi}$
$(sum1) \frac{SP_1 \vdash \varphi}{SP_1 \text{ and } SP_2 \vdash \iota_1(\varphi)}$	$(sum2) \frac{SP_2 \vdash \varphi}{SP_1 \text{ and } SP_2 \vdash \iota_2(\varphi)}$
$(trans) \frac{SP \vdash \varphi}{SP \text{ with } \sigma \vdash \sigma(\varphi)}$	$(derive) \frac{SP \vdash \sigma(\varphi)}{SP \text{ hide } \sigma \vdash \varphi}$

**Fig. 6.** Proof calculus for entailment in structured specifications

**Theorem 3.3 (Soundness [15]).** *The calculus for structured entailment is sound, i.e.  $SP \vdash \varphi$  implies  $SP \models \varphi$ . Also, the calculus for refinement between finite structured specifications is sound, i.e.  $SP_1 \rightsquigarrow SP_2$  implies  $SP_1 \approx SP_2$ .*

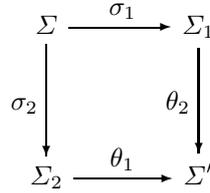
<sup>9</sup> Besides this model-theoretic notion of conservativeness, there also is a weaker consequence-theoretic notion:  $SP_2 \models \sigma(\varphi)$  implies  $SP_1 \models \varphi$ , and a proof-theoretic notion coinciding with the consequence-theoretic one for complete logics:  $SP_2 \vdash \sigma(\varphi)$  implies  $SP_1 \vdash \varphi$ . For the calculus of refinement, we need the model-theoretic notion.



**Fig. 7.** Proof calculus for refinement of structured specifications

Before we can state a completeness theorem, we need to formulate some technical assumptions on the underlying institution  $I$ .

An institution has the *Craig interpolation property* if for any pushout



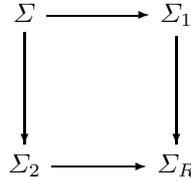
any  $\Sigma_1$ -sentence  $\varphi_1$  and any  $\Sigma_2$ -sentence  $\varphi_2$  with

$$\theta_2(\varphi_1) \models \theta_1(\varphi_2),$$

there exists a  $\Sigma$ -sentence  $\varphi$  (called the *interpolant*) such that

$$\varphi_1 \models \sigma_1(\varphi) \text{ and } \sigma_2(\varphi) \models \varphi_2.$$

A cocone for a diagram in **Sign** is called (*weakly*) *amalgamable* if it is mapped to a (weak) limit under **Mod**.  $\mathcal{I}$  (or **Mod**) admits (*finite*) (*weak*) *amalgamation* if (finite) colimit cocones are (weakly) amalgamable, i.e. if **Mod** maps (finite) colimits to (weak) limits. An important special case is pushouts in the signature category, which are prominently used for instance in instantiations of parameterized specifications, see Sect. 6.3. (Recall also that finite limits can be constructed from pullbacks and terminal objects, so that finite amalgamation reduces to preservation of pullbacks and terminal objects—dually: pushouts and initial objects). Here, the (weak) amalgamation property requires that a pushout



in **Sign** is mapped by **Mod** to a (weak) pullback

$$\begin{array}{ccc} \mathbf{Mod}(\Sigma) & \longleftarrow & \mathbf{Mod}(\Sigma_1) \\ \uparrow & & \uparrow \\ \mathbf{Mod}(\Sigma_2) & \longleftarrow & \mathbf{Mod}(\Sigma_R) \end{array}$$

of categories. Explicitly, this means that any pair  $(M_1, M_2) \in \mathbf{Mod}(\Sigma_1) \times \mathbf{Mod}(\Sigma_2)$  that is *compatible* in the sense that  $M_1$  and  $M_2$  reduce to the same  $\Sigma$ -model can be *amalgamated* to a unique (or weakly amalgamated to a not necessarily unique)  $\Sigma_R$ -model  $M$  (i.e., there exists a (unique)  $M \in \mathbf{Mod}(\Sigma_R)$  that reduces to  $M_1$  and  $M_2$ , respectively), and similarly for model morphisms.

An institution *has conjunction* if for any  $\Sigma$ -sentences  $\varphi_1$  and  $\varphi_2$ , there is a  $\Sigma$ -sentence  $\varphi$  that holds in a model iff  $\varphi_1$  and  $\varphi_2$  hold. The notion of an institution *having implication* is defined similarly.

**Theorem 3.4 (Completeness [15]).** *Under the assumptions that*

- *the institution has the Craig interpolation property,*
- *the institution admits weak amalgamation,*
- *the institution has conjunction and implication and*
- *the logic is complete,*

*the calculi for structured entailment and refinement between finite structured specifications are complete.*

Actually, the assumption of Craig interpolation and weak amalgamation can be restricted to those diagrams for which it is really needed. Details can be found in [15].

Notice though that even a stronger version of the interpolation property, namely Craig-Robinson interpolation as in [20], still needs closure of the set of sentences under implication in order to ensure the completeness of the above compositional proof system.

A problem with the above result is that Craig interpolation often fails, e.g. it does not hold for the CASL institution  $SubPCFOL^=$  (although it does hold for the sublanguage with sort-injective signature morphisms and without subsorts and sort generation constraints, see [13]). This problem may be overcome by adding a “global” rule to the calculus, which does a kind of normal form computation, while maintaining the structure of specifications to guide proof search as much as possible; see [41, 42].

*Checking conservativity in the CASL institution*

The proof rules for refinement are based on an oracle that checks conservativeness of extensions. Hence, logic-specific rules for checking conservativeness are needed. For CASL, conservativeness can be checked by syntactic criteria:

e.g. free types and recursive definitions over them are always conservative. But more sophisticated rules are also available, see [44]. Note that checking conservativeness is at least as complicated as checking non-provability: for a  $\Sigma$ -specification  $SP$ ,  $SP \not\models \varphi$  iff  $SP$  **and**  $\langle \Sigma, \{\neg\varphi\} \rangle$  is consistent iff  $SP$  **and**  $\langle \Sigma, \{\neg\varphi\} \rangle$  is conservative over the empty specification. Hence, even checking conservativeness in first-order logic is not recursively enumerable, and thus there is no recursively axiomatized complete calculus for this task.<sup>10</sup>

#### *Proof rules for free specifications*

An institution independent proof theory for free specifications has not been developed yet (if this should be feasible at all). Hence, for free specifications, one needs to develop proof support for each institution separately. For the CASL institution, this has been sketched in [40]. The main idea is just to mimic a quotient term algebra construction, and to restrict proof support to those cases (e.g. Horn clause theories) where the free model is given by such a construction. Details can be found in [40].

### 6.3 Named and Parameterized Specifications and Views

Structured specifications may be *named*, so that the reuse of a specification may be replaced by a *reference* to it through its name. A named specification may declare some *parameters*, the union of which is extended by a *body*; it is then called *generic*. This is written as

$$\mathbf{spec} \text{ } SpName[ParSp] = BodySp$$

where  $BodySp$  is an extension of  $ParSp$ . See Fig. 3 for an example of a generic specification of lists.

A reference to a generic specification should *instantiate* it by providing, for each parameter, an *argument specification* together with a *fitting morphism* from the parameter to the argument specification. Fitting may also be achieved by (explicit) use of named *views* between the parameter and argument specifications. The union of the arguments, together with the translation of the generic specification by an expansion of the fitting morphism, corresponds to a pushout construction—taking into account any explicit *imports* of the generic specification, which allow symbols used in the body to be declared also by arguments.

<sup>10</sup> The situation is even more subtle. The model-theoretic notion of conservative extension (or, equivalently, of refinement between specifications involving hiding) corresponds to second-order existential quantification. It is well-known that the semantics of second-order logic depends on the background set theory [32]. For example, one can build a specification and its extension that is conservative (or equivalently, provide another specification to which it refines) iff the continuum hypothesis holds—a question that is independent of our background metatheory *ZFCU*.

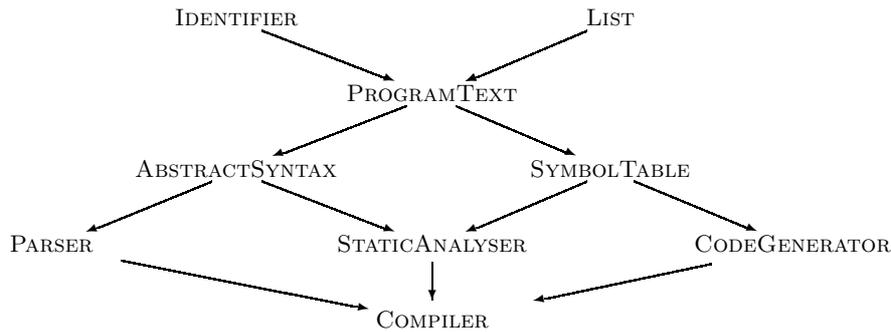
Since parameterization may be expressed in terms of union and translation, we omit its semantics and proof rules here.

Semantically, a view  $v: SP_1 \rightarrow SP_2$  from a  $\Sigma_1$ -specification  $SP_1$  to a  $\Sigma_2$ -specification  $SP_2$  is basically a *specification morphism*  $\sigma: SP_1 \rightarrow SP_2$ , leading to a proof obligation  $SP_1 \rightsquigarrow SP_2 \text{ hide } \sigma$ . A similar proof obligation is generated for anonymous instantiations of parameterized specifications (i.e. not given by a named view).

Naming specifications and referencing them by name leads to *graphs* of specifications. This is formalized as a so-called *development graph* [41, 42, 44], which express *sharing* between specifications, thereby leading to a more efficient proof calculus, and providing management of proof obligations and proofs for structured specification, as well as management of change.

## 7 Architectural Specifications

Architectural specifications in CASL provide a means of stating how implementation units are used as building blocks for larger components. (Dynamic interaction between modules and dynamic changes of software structure are currently beyond the scope of this approach.)



**Fig. 8.** Structure of the specification of a compiler. The arrows indicate the extension relation between specifications

Units are represented as names to which a specification is associated. Such a named unit is to be thought of as an arbitrarily selected model of the specification. Units may be parameterized, where specifications are associated with both the parameters and the result. The result specification is required to extend the parameter specifications. A parameterized unit is to be understood as a function which, given models of the parameter specifications, outputs a model of the result specification; this function is required to be *persistent* in

the sense that reducing the result to the parameter signatures reproduces the parameters.

Units can be assembled via unit expressions which may contain operations such as renaming or hiding of symbols, amalgamation of units, and application of a parameterized unit. Terms containing such operations will only be defined if symbols that are identified, e.g. by renaming them to the same symbol or by amalgamating units that have symbols in common, are also interpreted in the same way in all “collective” models of the units defined so far.

An architectural specification consists of declarations and/or definitions of a number of units, together with a way of assembling them to yield a result unit.

*Example 3.5.* A (fictitious) specification structure for a compiler might look roughly as depicted in Fig. 8. The corresponding architectural specification in CASL might have the following form:

```

arch spec BUILDCOMPILER =
units I : IDENTIFIER with sorts Identifier, Keyword;
      L : ELEM → LIST[ELEM];
      IL = L[I fit sort Elem ↦ Identifier]
      KL = L[I fit sort Elem ↦ Keyword]
      PT : PROGRAMTEXT given IL, KL;
      AS : ABSTRACTSYNTAX given PT;
      ST : SYMBOLTABLE given PT;
      P : PARSER given AS;
      SA : STATICANALYSER given AS, ST;
      CG : CODEGENERATOR given ST
result P and SA and CG
end

```

(Here, the keyword **with** is used just to list some of the defined symbols. The keyword **given** indicates imports.) According to the above specification, the parser, the static analyser, and the code generator would be constructed building upon a given abstract syntax and a given mechanism for symbol tables, and the compiler would be obtained by just putting together the former three units. Roughly speaking, this is only possible (in a manner that can be statically checked) if all symbols that are shared between the parser, the static analyser and the code generator already appear in the units for the abstract syntax or the symbol tables—otherwise, incompatibilities might occur that make it impossible to put the separately developed components together. For instance, if both `STATICANALYSER` and `CODEGENERATOR` declare an operation *lookup* that serves to retrieve symbols from the symbol table, then the corresponding implementations might turn out to be substantially different, so that the two components fail to be compatible. Of course, this points

to an obvious flaw in the architecture: *lookup* should have been declared in SYMBOLTABLE.

Consider an institution with unions  $I = (\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \models)$ . We assume that the signature category is finitely cocomplete and that the institution admits amalgamation. We also assume that signature unions are exhaustive in the sense that given two signatures  $\Sigma_1$  and  $\Sigma_2$  and their union  $\Sigma_1 \xrightarrow{\iota_1} (\Sigma_1 \cup \Sigma_2) \xleftarrow{\iota_2} \Sigma_2$ , for any models  $M_1 \in \mathbf{Mod}(\Sigma_1)$  and  $M_2 \in \mathbf{Mod}(\Sigma_2)$ , there is at most one model  $M \in \mathbf{Mod}(\Sigma_1 \cup \Sigma_2)$  such that  $M|_{\iota_1} = M_1$  and  $M|_{\iota_2} = M_2$ . In such a framework we formally present a small but representative subset of CASL architectural specifications. The fragment—or rather, its syntax—is given in Fig. 9.

Architectural specifications:  $ASP ::= \mathbf{arch\ spec\ } Dcl^* \mathbf{\ result\ } T$   
 Unit declarations:  $Dcl ::= U : SP \mid U : SP_1 \xrightarrow{\tau} SP_2$   
 Unit terms:  $T ::= U \mid U[T \mathbf{\ fit\ } \sigma] \mid T_1 \mathbf{\ and\ } T_2$

**Fig. 9.** A fragment of the architectural specification formalism

Example 3.5 additionally uses unit definitions and imports. Unit definitions  $U = T$  introduce a (non-parameterized) unit and give its value by a unit term. Imports can be regarded as syntactical sugar for a parameterized unit which is instantiated only once: if  $U_1 : \mathbf{SPEC}_1$ , then

$$U_2 : \mathbf{SPEC}_2 \mathbf{\ given\ } U_1$$

abbreviates

$$\begin{aligned} U'_2 &: \mathbf{SPEC}_1 \rightarrow \mathbf{SPEC}_2; \\ U_2 &= U'_2[U_1]. \end{aligned}$$

We now sketch the formal semantics of our language fragment and show how correctness of such specifications may be established.

### 7.1 Semantics of Architectural Specifications

The semantics of architectural specifications introduced above is split into their static and model semantics, in very much the same way as for structured specifications in Sect. 6.

Unit terms are statically elaborated in a *static context*  $C_{st} = (P_{st}, B_{st})$ , where  $P_{st}$  maps parameterized unit names to signature morphisms and  $B_{st}$  maps non-parameterized unit names to their signatures. We require the domains of  $P_{st}$  and  $B_{st}$  to be disjoint. The empty static context that consists of two empty maps will be written as  $C_{st}^\emptyset$ . Given an initial static context, the static semantics for unit declarations produces a static context by adding the signature for the newly introduced unit, and the static semantics for unit terms determines the signature for the resulting unit.

$$\boxed{
\begin{array}{c}
\frac{\vdash UDD^* \triangleright C_{st} \quad C_{st} \vdash T \triangleright \Sigma}{\vdash \mathbf{arch\ spec\ } UDD^* \mathbf{ result\ } T \triangleright (C_{st}, \Sigma)} \\
\\
\frac{C_{st}^\emptyset \vdash UDD_1 \triangleright (C_{st})_1 \quad \cdots \quad (C_{st})_{n-1} \vdash UDD_n \triangleright (C_{st})_n}{\vdash UDD_1 \dots UDD_n \triangleright (C_{st})_n} \\
\\
\frac{\vdash SP \triangleright \Sigma \quad U \notin (Dom(P_{st}) \cup Dom(B_{st}))}{(P_{st}, B_{st}) \vdash U : SP \triangleright (P_{st}, B_{st} + \{U \mapsto \Sigma\})} \\
\\
\frac{\vdash SP_1 \triangleright \Sigma_1 \quad \vdash SP_2 \triangleright \Sigma_2 \quad \tau : \Sigma_1 \rightarrow \Sigma_2 \quad U \notin (Dom(P_{st}) \cup Dom(B_{st}))}{(P_{st}, B_{st}) \vdash U : SP_1 \xrightarrow{\tau} SP_2 \triangleright (P_{st} + \{U \mapsto \tau\}, B_{st})} \\
\\
\frac{U \in Dom(B_{st})}{(P_{st}, B_{st}) \vdash U \triangleright B_{st}(U)} \\
\\
\frac{P_{st}(U) = \tau : \Sigma \rightarrow \Sigma' \quad C_{st} \vdash T \triangleright \Sigma_T \quad \sigma : \Sigma \rightarrow \Sigma_T \quad (\tau' : \Sigma_T \rightarrow \Sigma'_T, \sigma' : \Sigma' \rightarrow \Sigma'_T \text{ is the pushout of } (\sigma, \tau))}{(P_{st}, B_{st}) \vdash U[T \mathbf{fit} \sigma] \triangleright \Sigma'_T} \\
\\
\frac{\Sigma = \Sigma_1 \cup \Sigma_2 \text{ with inclusions } \iota_1 : \Sigma_1 \rightarrow \Sigma, \iota_2 : \Sigma_2 \rightarrow \Sigma \quad C_{st} \vdash T_1 \triangleright \Sigma_1 \quad C_{st} \vdash T_2 \triangleright \Sigma_2}{(P_{st}, B_{st}) \vdash T_1 \mathbf{and} T_2 \triangleright \Sigma}
\end{array}
}$$

Fig. 10. Static semantics of architectural specifications

In terms of the model semantics, a (non-parameterized) unit  $M$  over a signature  $\Sigma$  is just a model  $M \in \mathbf{Mod}(\Sigma)$ . A parameterized unit  $F$  over a parameterized unit signature  $\tau : \Sigma_1 \rightarrow \Sigma_2$  is a persistent partial function  $F : \mathbf{Mod}(\Sigma_1) \rightarrow \mathbf{Mod}(\Sigma_2)$  (i.e.  $F(M)|_\tau = M$  for each  $M \in Dom(F)$ ).

The model semantics for architectural specifications involves interpretations of unit names. These are given by *unit environments*  $E$ , i.e. finite maps from unit names to units as introduced above. On the model semantics side, the analogue of a static context is a *unit context*  $\mathcal{C}$ , which is just a class of unit environments, and can be thought of as a constraint on the interpretation of unit names. The unconstrained unit context, which consists of all environments, will be written as  $\mathcal{C}^\emptyset$ . The model semantics for unit declarations modifies unit contexts by constraining the environments to interpret the newly introduced unit names as determined by their specification or definition.

A unit term is interpreted by a *unit evaluator*  $UEv$ , a function that yields a unit when given a unit environment in the unit context (the unit environment serves to interpret the unit names occurring in the unit term). Hence, the model semantics for a unit term yields a unit evaluator, given a unit context.

The complete semantics is given in Figs. 10 (static semantics) and 11 (model semantics) where we use some auxiliary notation: given a unit context

$$\begin{array}{c}
\frac{\vdash UDD^* \Rightarrow \mathcal{C} \quad \mathcal{C} \vdash T \Rightarrow UEv}{\vdash \text{arch spec } UDD^* \text{ result } T \Rightarrow (\mathcal{C}, UEv)} \\
\\
\frac{\mathcal{C}^0 \vdash UDD_1 \Rightarrow \mathcal{C}_1 \quad \cdots \quad \mathcal{C}_{n-1} \vdash UDD_n \Rightarrow \mathcal{C}_n}{\vdash UDD_1 \dots UDD_n \Rightarrow \mathcal{C}_n} \\
\\
\frac{\vdash SP \Rightarrow \mathcal{M}}{\mathcal{C} \vdash U : SP \Rightarrow \mathcal{C} \times \{U \mapsto \mathcal{M}\}} \\
\\
\frac{\vdash SP_1 \Rightarrow \mathcal{M}_1 \quad \vdash SP_2 \Rightarrow \mathcal{M}_2 \quad \mathcal{F} = \{F : \mathcal{M}_1 \rightarrow \mathcal{M}_2 \mid \text{for } M \in \mathcal{M}_1, F(M)|_{\tau} = M\}}{\mathcal{C} \vdash U : SP_1 \xrightarrow{\tau} SP_2 \Rightarrow \mathcal{C} \times \{U \mapsto \mathcal{F}\}} \\
\\
\frac{}{\mathcal{C} \vdash U \Rightarrow \{E \mapsto E(U) \mid E \in \mathcal{C}\}} \\
\\
\frac{\mathcal{C} \vdash T \Rightarrow UEv \quad \left. \begin{array}{l} \text{for each } E \in \mathcal{C}, UEv(E)|_{\sigma} \in \text{Dom}(E(U)) \end{array} \right\} (*)}{\left. \begin{array}{l} \text{for each } E \in \mathcal{C}, \text{ there is a unique } M \in \mathbf{Mod}(\Sigma'_T) \text{ such that} \\ M|_{\tau'} = UEv(E) \text{ and } M|_{\sigma'} = E(U)(UEv(E)|_{\sigma}) \end{array} \right\} (**)}{UEv' = \{E \mapsto M \mid E \in \mathcal{C}, M|_{\tau'} = UEv(E), M|_{\sigma'} = E(U)(UEv(E)|_{\sigma})\}} \\
\frac{}{\mathcal{C} \vdash U[T \text{ fit } \sigma] \Rightarrow UEv'} \\
\\
\frac{\mathcal{C} \vdash T_1 \Rightarrow UEv_1 \quad \mathcal{C} \vdash T_2 \Rightarrow UEv_2 \quad \left. \begin{array}{l} \text{for each } E \in \mathcal{C}, \text{ there is a unique } M \in \mathbf{Mod}(\Sigma) \text{ such that} \\ M|_{\iota_1} = UEv_1(E) \text{ and } M|_{\iota_2} = UEv_2(E) \end{array} \right\} (***)}{UEv = \{E \mapsto M \mid E \in \mathcal{C} \text{ and } M|_{\iota_1} = UEv_1(E), M|_{\iota_2} = UEv_2(E)\}} \\
\frac{}{\mathcal{C} \vdash T_1 \text{ and } T_2 \Rightarrow UEv}
\end{array}$$

Fig. 11. Model semantics of architectural specifications

$\mathcal{C}$ , a unit name  $U$  and a class  $\mathcal{V}$ ,

$$\mathcal{C} \times \{U \mapsto \mathcal{V}\} := \{E + \{U \mapsto V\} \mid E \in \mathcal{C}, V \in \mathcal{V}\},$$

where  $E + \{U \mapsto V\}$  maps  $U$  to  $V$  and otherwise behaves like  $E$ . The model semantics assumes that the static semantics has been successful on the constructs considered; we use the notations introduced by this derivation of the static semantics in the model semantics rules whenever convenient.

The model semantics is easily seen to be compatible with the static semantics in the following sense: we say that  $\mathcal{C}$  fits  $C_{st} = (P_{st}, B_{st})$ , if, whenever  $B_{st}(U) = \Sigma$  and  $E \in \mathcal{C}$ , then  $E(U)$  is a  $\Sigma$ -model, and a corresponding condition holds for  $P_{st}$ . Obviously,  $\mathcal{C}^0$  fits  $C_{st}^0$ . Now if  $\mathcal{C}$  fits  $C_{st}$ , then  $C_{st} \vdash T \triangleright \Sigma$  and  $\mathcal{C} \vdash T \Rightarrow UEv$  imply that  $UEv(E)$  is a  $\Sigma$ -model for each  $E \in \mathcal{C}$ . Corresponding statements hold for the other syntactic categories (unit declarations, architectural specifications).

We say that an architectural specification is internally correct (or simply: *correct*) if it has both static and model semantics. Informally, this means that the architectural design the specification captures is correct in the sense that any realization of the units according to their specifications allows us to construct an overall result by performing the construction prescribed by the result unit term.

Checking correctness of an architectural specification requires checking that all the rules necessary for derivation of its semantics may indeed be applied, that is, all their premises can be derived and the conditions they capture hold. Perhaps the only steps which require further discussion are the rules of the model semantics for unit application and amalgamation in Fig. 11. Only there do some difficult premises occur, marked by (\*), (\*\*) and (\*\*\*), respectively. All the other premises of the semantic rules are “easy” in the sense that they largely just pass on the information collected about various parts of the given phrase, or perform a very simple check that names are introduced before being used, signatures fit as expected, etc.

First we consider the premises (\*\*) and (\*\*\*) in the rules for unit application and amalgamation, respectively. They impose “amalgamability requirements”, necessary to actually build the expected models by combining the simpler models, as indicated. Such requirements are typically expected to be at least partially discharged by static analysis—similarly to the sharing requirements present in some programming languages (cf. e.g. *Standard ML* [53]). Under our assumptions, the premise (\*\*) may simply be skipped, as it always holds (since all parameterized units are persistent functions,  $E(U)(UEv(E)|_\sigma)|_\tau = UEv(E)|_\sigma$ , and so the required unique model  $M \in \mathbf{Mod}(\Sigma'_T)$  exists by the amalgamation property of the institution). The premise (\*\*\*) may fail though, and a more subtle static analysis of the dependencies between units may be needed to check that it holds for a given construct.

The premise (\*) in the rule for application of a parameterized unit requires that the fitting morphism correctly “fits” the actual parameter as an argument for the parameterized unit. To verify this, one typically has to prove that the fitting morphism is a specification morphism from the argument specification to the specification of the actual parameter. Similarly as with the proof obligations arising for instantiations of parameterized specifications discussed in Sect. 6.3, this in general requires some semantic or proof-theoretic reasoning. Moreover, a suitable calculus is needed to determine a specification for the actual parameter. One naive attempt to provide it might be to build such a specification inductively for each unit term using directly specifications of its components. Let  $SP_T$  be such a specification for the term  $T$ . In other words, verification conditions aside:

- $SP_U$  is  $SP$ , where  $U: SP$  is the declaration of  $U$ ;
- $SP_{T_1 \text{ and } T_2}$  is  $(SP_{T_1} \text{ and } SP_{T_2})$ ;

- $SP_{U[T \text{ fit } \sigma]}$  is  $((SP_T \text{ with } \tau') \text{ and } (SP' \text{ with } \sigma'))$ , where  $U: SP \xrightarrow{\tau} SP'$  is the declaration of  $U$  and  $(\tau', \sigma')$  is the pushout of  $(\sigma, \tau)$ , as in the corresponding rule of the static semantics.

It can easily be seen that  $SP_T$  so determined is indeed a correct specification for  $T$ , in the sense that if  $C_{st} \vdash T \triangleright \Sigma$  and  $\mathcal{C} \vdash T \Rightarrow UEv$  then  $\vdash SP_T \triangleright \Sigma$  and  $\vdash SP_T \Rightarrow \mathcal{M}$  with  $UEv(E) \in \mathcal{M}$  for each  $E \in \mathcal{C}$ . Therefore, we could replace the requirement  $(*)$  by  $SP \approx SP_T \text{ hide } \sigma$ .

However, this would be highly incomplete. Consider a trivial example:

```

units  $U : \{\text{sort } s; \text{op } a : s\}$ 
       $ID : \{\text{sort } s; \text{op } b : s\} \rightarrow \{\text{sort } s; \text{op } b : s\}$ 
       $F : \{\text{sort } s; \text{op } a, b : s; \text{axiom } a = b\} \rightarrow \dots$ 
result  $F[ U \text{ and } ID[U \text{ fit } b \mapsto a] ]$ 

```

The specification we obtain for the argument unit term of  $F$  does not capture that fact that  $a = b$  holds in all units that may actually arise as the argument for  $F$  here. The problem is that the specification for a unit term built as above entirely disregards any dependencies and sharing that may occur between units denoted by unit terms, and so is often insufficient to verify correctness of unit applications. Hence, this first try to calculate specifications for architectural unit terms turns out to be inadequate, and a more complex form of architectural verification is needed.

## 7.2 Verification

The basic idea behind verification for architectural specifications is that we want to extend the static information about units to capture their properties by an additional specification. However, as discussed at the end of the previous section, we must also take into account sharing between various unit components, resulting from inheritance of some unit parts via for instance parameterized unit applications. To capture this, we accumulate information on non-parameterized units in a single *global signature*  $\Sigma_G$ , and represent non-parameterized unit signatures as morphisms into this global signature, assigning them to unit names by a map  $B_v$ . The additional information resulting from the unit specifications will be accumulated in a single *global specification*  $SP_G$  over this signature (i.e., we will always have  $\vdash SP_G \triangleright \Sigma_G$ ). Finally, we will of course store the entire specification for each parameterized unit, assigning them to parameterized unit names by a map  $P_v$ . This results in the concept of a *verification context*  $C_v = (P_v, B_v, SP_G)$ . A static unit context  $ctx(C_v) = (P_{st}, B_{st})$  may easily be extracted from such an extended one: for each  $U \in \text{Dom}(B_v)$ ,  $B_{st}(U) = \Sigma$ , where  $B_v(U) = i : \Sigma \rightarrow \Sigma_G$ , and for each  $U \in \text{Dom}(P_v)$ ,  $P_{st}(U) = \tau$ , where  $P_v(U) = SP_1 \xrightarrow{\tau} SP_2$ .

Given a morphism  $\theta: \Sigma_G \rightarrow \Sigma'_G$  that extends the global signature (or a global specification morphism  $\theta: SP_G \rightarrow SP'_G$ ) we write  $B_v; \theta$  for the corre-

$$\begin{array}{c}
\frac{\vdash Dcl^* :: C_v \quad C_v \vdash T :: \Sigma \xrightarrow{i} SP'_G \xleftarrow{\theta} SP_G}{\vdash \mathbf{arch\ spec\ } Dcl^* \mathbf{ result\ } T :: SP'_G \mathbf{ hide\ } i} \\
\\
\frac{C_v^\emptyset \vdash Dcl_1 :: (C_v)_1 \quad \cdots \quad (C_v)_{n-1} \vdash Dcl_n :: (C_v)_n}{\vdash Dcl_1 \dots Dcl_n :: (C_v)_n} \\
\\
\frac{\begin{array}{c} U \notin (Dom(P_v) \cup Dom(B_v)) \quad \vdash SP \triangleright \Sigma \\ (\Sigma_G \xrightarrow{\theta} \Sigma'_G \xleftarrow{i} \Sigma) \text{ is the coproduct of } \Sigma_G \text{ and } \Sigma \end{array}}{(P_v, B_v, SP_G) \vdash U : SP :: \\ (P_v, (B_v; \theta) + \{U \mapsto i\}, (SP_G \mathbf{ with\ } \theta) \mathbf{ and\ } (SP \mathbf{ with\ } i))} \\
\\
\frac{U \notin (Dom(P_v) \cup Dom(B_v))}{(P_v, B_v, SP_G) \vdash U : SP_1 \xrightarrow{\tau} SP_2 :: (P_v + \{U \mapsto SP_1 \xrightarrow{\tau} SP_2\}, B_v, SP_G)} \\
\\
\frac{B_v(U) = \Sigma \xrightarrow{i} SP_G}{(P_v, B_v, SP_G) \vdash U :: \Sigma \xrightarrow{i} SP_G \xleftarrow{id} SP_G} \\
\\
\frac{\begin{array}{c} (P_v, B_v, SP_G) \vdash T :: \Sigma_T \xrightarrow{i} SP'_G \xleftarrow{\theta} SP_G \\ P_v(U) = SP \xrightarrow{\tau} SP' \quad \vdash SP \triangleright \Sigma \quad \vdash SP' \triangleright \Sigma' \quad \sigma : \Sigma \rightarrow \Sigma_T \\ (\tau' : \Sigma_T \rightarrow \Sigma'_T, \sigma' : \Sigma' \rightarrow \Sigma'_T) \text{ is the pushout of } (\sigma, \tau) \\ (\tau'' : \Sigma'_G \rightarrow \Sigma''_G, i' : \Sigma'_T \rightarrow \Sigma''_G) \text{ is the pushout of } (i, \tau') \\ SP \mathbf{ with\ } \sigma; i \rightsquigarrow SP'_G \end{array}}{(P_v, B_v, SP_G) \vdash U[T \mathbf{ fit\ } \sigma] :: \\ \Sigma'_T \xrightarrow{i'} (SP'_G \mathbf{ with\ } \tau'') \mathbf{ and\ } (SP' \mathbf{ with\ } \sigma'; i') \xleftarrow{\theta; \tau''} SP_G} \\
\\
\frac{\begin{array}{c} (P_v, B_v, SP_G) \vdash T_1 :: \Sigma_1 \xrightarrow{i_1} SP_G^1 \xleftarrow{\theta_1} SP_G \\ (P_v, B_v, SP_G) \vdash T_2 :: \Sigma_2 \xrightarrow{i_2} SP_G^2 \xleftarrow{\theta_2} SP_G \\ \Sigma = \Sigma_1 \cup \Sigma_2 \text{ with inclusions } \iota_1 : \Sigma_1 \rightarrow \Sigma, \iota_2 : \Sigma_2 \rightarrow \Sigma \\ (\theta'_2 : \Sigma_G^1 \rightarrow \Sigma'_G, \theta'_1 : \Sigma_G^2 \rightarrow \Sigma'_G) \text{ is the pushout of } (\theta_1, \theta_2) \\ j : \Sigma \rightarrow \Sigma'_G \text{ satisfies } \iota_1; j = i_1; \theta'_2 \text{ and } \iota_2; j = i_2; \theta'_1 \end{array}}{(P_v, B_v, SP_G) \vdash T_1 \mathbf{ and\ } T_2 :: \\ \Sigma_1 \cup \Sigma_2 \xrightarrow{j} (SP_G^1 \mathbf{ with\ } \theta'_2) \mathbf{ and\ } (SP_G^2 \mathbf{ with\ } \theta'_1) \xleftarrow{\theta'_1; \theta'_2} SP_G}
\end{array}$$

Fig. 12. Verification rules

sponding extension of  $B_v$  (mapping each  $U \in Dom(B_v)$  to  $B_v(U; \theta)$ ).  $C_v^\emptyset$  is the “empty” verification context (with the initial global specification<sup>11</sup>).

The intuition introduced above is reflected in the forms of verification judgments, and captured formally by the verification rules:

$$\boxed{\vdash ASP :: SP}$$

<sup>11</sup> More precisely, this is the basic specification consisting of the initial signature with no axioms.

Architectural specifications yield a specification of the result.

$$\boxed{C_v \vdash Decl :: C'_v}$$

In a verification context, unit declarations yield a new verification context.

$$\boxed{(P_v, B_v, SP_G) \vdash T :: \Sigma \xrightarrow{i} SP'_G \xleftarrow{\theta} SP_G}$$

In a verification context, unit terms yield their signature embedded into a new global specification, obtained as an indicated extension of the old global specification.

The verification rules to derive these judgments are in Fig. 12, with diagrams helping to read the more complicated rules for unit application and amalgamation in Fig. 13.

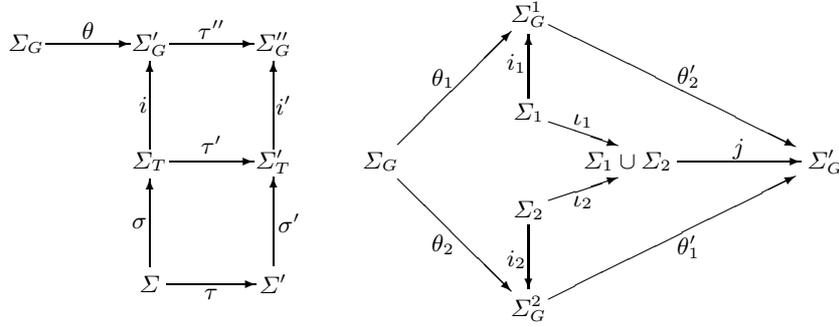


Fig. 13. Diagrams for unit application and amalgamation

It should be easy to see that the verification semantics subsumes (in the obvious sense) the static semantics: a successful derivation of the verification semantics ensures a successful derivation of the static semantics with results that may be extracted from the results of the verification semantics in the obvious way.

More crucially, a successful derivation of the verification semantics on an architectural specification ensures a successful derivation of the model semantics, and hence the correctness of the architectural specification.

To state this more precisely, we need an extension to verification contexts of the notion that a unit context fits a static context: a unit context  $C_v$  fits a verification context  $C_v = (P_v, B_v, SP_G)$ , where  $\vdash SP_G \Rightarrow \mathcal{M}_G$ , if

- for each  $E \in \mathcal{C}$  and  $U \in \text{Dom}(P_v)$  with  $P_v(U) = SP \xrightarrow{\tau} SP'$ , where  $\vdash SP \Rightarrow \mathcal{M}$  and  $\vdash SP' \Rightarrow \mathcal{M}'$ , we have  $E(U)(M) \in \mathcal{M}'$  for all  $M \in \mathcal{M}$ , and
- for each  $E \in \mathcal{C}$ , there exists  $M_G \in \mathcal{M}_G$  such that for all  $U \in \text{Dom}(B_v)$ ,  $E(U) = M_G|_{B_v(U)}$ ; we say then that  $E$  is witnessed by  $M_G$ .

Now, the following claims follow by induction:

- For every architectural specification  $ASP$ , if  $\vdash ASP :: SP$  with  $\vdash SP \Rightarrow \mathcal{M}$  then  $\vdash ASP \Rightarrow (\mathcal{C}, UEv)$  for some unit context  $\mathcal{C}$  and unit evaluator  $UEv$  such that  $UEv(E) \in \mathcal{M}$  for all  $E \in \mathcal{C}$ .
- For any unit declaration  $Dcl$  and verification context  $C_v$ , if  $C_v \vdash Dcl :: C'_v$  then for any unit context  $\mathcal{C}$  that fits  $C_v$ ,  $\mathcal{C} \vdash Dcl \Rightarrow \mathcal{C}'$  for some unit context  $\mathcal{C}'$  that fits  $C'_v$ ; this generalizes to sequences of unit declarations in the obvious way.
- For any unit term  $T$  and verification context  $C_v = (P_v, B_v, SP_G)$ , where  $\vdash SP_G \Rightarrow \mathcal{M}_G$ , if  $C_v \vdash T :: \Sigma \xrightarrow{i} SP'_G \xleftarrow{\theta} SP_G$ , where  $\vdash SP'_G \Rightarrow \mathcal{M}'_G$ , then for any unit context  $\mathcal{C}$  that fits  $C_v$ ,  $\mathcal{C} \vdash T \Rightarrow UEv$  for some unit evaluator  $UEv$  such that for each  $E \in \mathcal{C}$  witnessed by  $M_G \in \mathcal{M}_G$ , there exists a model  $M'_G \in \mathcal{M}'_G$  such that  $M'_G|_{\theta} = M_G$  and  $M'_G|_i = UEv(E)$ .

In particular this means that a successful derivation of the verification semantics ensures that in the corresponding derivation of the model semantics, whenever the rules for unit application and amalgamation are invoked, the premises marked by (\*), (\*\*) and (\*\*\*) hold. This may also be seen somewhat more directly:

- (\*) Given the above relationship between verification and model semantics, the requirement (\*) in the model semantics rule for unit application follows from the requirement that  $SP$  **with**  $\sigma; i \rightsquigarrow SP'_G$  in the corresponding verification rule.
- (\*\*) As pointed out already, the premises marked by (\*\*) may be removed by the assumption that the institution we work with admits amalgamation.
- (\*\*\*) Given the above relationship between verification and model semantics, the existence of models required by (\*\*\*) in the model semantics rule for unit amalgamation can be shown by gradually constructing a compatible family of models over the signatures in the corresponding diagram in Fig. 13 (this requires amalgamation again); the uniqueness of the model so constructed follows from our assumption on signature union.

Note that only checking the requirement (\*) relies on the information gathered in the specifications built for unit terms by the verification semantics. The other requirements are entirely “static” in the sense that they may be checked also if we replace specifications by their signatures. This may be used to further split the verification semantics into two parts: an extended static analysis, performed without taking specifications into account, but considering in detail all the mutual dependencies between units involved to check properties like those labeled by (\*\*) and (\*\*\*); and a proper verification semantics aimed at considering unit specifications and deriving specifications for unit terms from them. See [67] for details.

### 7.3 Enriched CASL, Diagram Semantics and the Cell Calculus

The verification semantics of architectural specifications presented in the previous section crucially depends on amalgamation in the underlying institution. However, the CASL institution fails to have this property:

*Example 3.6.* The simplest case where amalgamation fails is the following: let  $\Sigma$  be the signature with sorts  $s$  and  $t$  and no operations, and let  $\Sigma_1$  be the extension of  $\Sigma$  by the subsort relation  $s \leq t$ . Then the pushout

$$\begin{array}{ccc} \Sigma & \longrightarrow & \Sigma_1 \\ \downarrow & & \downarrow \\ \Sigma_1 & \longrightarrow & \Sigma_1 \end{array}$$

in **SubSig** fails to be amalgamable (since two models of  $\Sigma_1$  that are compatible w.r.t. the inclusion of  $\Sigma$  may interpret the subsort injection differently).

The solution is to embed the CASL institution into an institution that enjoys the amalgamation property. The main idea in the definition of the required extended institution is to generalize pre-orders of sorts to *categories* of sorts, i.e. to admit several different subsort embeddings between two given sorts; this gives rise to the notion of *enriched CASL signature*. Details can be found in [63]. This means that before a CASL architectural specification can be statically checked and verification conditions can be proved, it has to be translated to enriched CASL, using the embedding.

One might wonder why the mapping from subsorted to many-sorted specifications introduced in Sect. 4 is not used instead of introducing enriched CASL. Indeed, this is possible. However, enriched CASL has the advantage of keeping the subsorting information entirely static, avoiding any axioms to capture the built-in structural properties, as would be the case with the mapping from Sect. 4.

This advantage plays a role in the so-called *diagram semantics* of architectural specifications. It replaces the global signatures that are used in the static semantics by diagrams of signatures and signature morphisms, see [9]. In the “extended static part” of the verification semantics, the commutativity conditions concerning signature morphisms into the global signature have then to be replaced by model-theoretic amalgamation conditions. Given an embedding into an institution with amalgamation such as the one discussed above, the latter conditions are equivalent to factorization conditions of the colimit of the embedded diagram. For (enriched) CASL, these factorization conditions can be dealt with using a calculus (the so-called *cell calculus*) for proving equality of morphisms and symbols in the colimit; see [29]. A verification semantics without reference to overall global specifications (which relies on the amalgamation property) and consequently with more “local” verification conditions is yet to be worked out.

## 8 Refinement

The standard development paradigm of algebraic specification [5] postulates that formal software development begins with a formal *requirement specification* (extracted from a software project's informal requirements) that fixes only expected properties but ideally says nothing about implementation issues; this is to be followed by a number of *refinement* steps that fix more and more details of the design, so that one finally arrives at what is often termed the *design specification*. The last refinement step then results in an actual *implementation* in a programming language.

One aspect of refinement concerns the way that the specified model class gets smaller and smaller as more and more design decision are made during the refinement process, until a monomorphic design specification or program is reached. This is reflected by CASL's concepts of *views* and the corresponding refinement relation  $\approx\approx$  between specifications as introduced in Sect. 6. However, views are not expressive enough for refinement, being primarily a means for naming fitting morphisms for parameterized specifications. This is because there are more aspects of refinement than just model class inclusion.

One central issue here is so-called *constructor refinement* [61]. This includes the basic constructions for writing implementation units that can be found in programming languages, e.g. enumeration types, algebraic datatypes (that is, free types) and recursive definitions of operations. Also, unit terms in architectural specifications can be thought of as (logic independent) constructors: they construct larger units out of smaller ones. Refinements may use these constructors, and hence the task of implementing a specification may be entirely discharged (by supplying appropriate constructs in some programming language), or may be reduced (via an architectural specification) to the implementation of smaller specifications. A first refinement language following these lines is described in [46]. In this language, one can on the one hand express chains of model class inclusions, like

```

refinement R1 =
  SP1 refined to
    SP2 refined to SP3
end

```

expressing that the model class of SP3 is included in that of SP2, which is in turn included in the model class of SP1. On the other hand, it is possible to refine structured specifications to architectural specifications, introducing a branching into the development:

```

refinement R2 =
  SP1 refined to arch spec ASP
end

```

Architectural specifications can be further refined by refining their components, like in:

```

refinement R3 =
  SP refined to arch spec units
     $K : SP' \rightarrow SP$ 
     $A' : SP'$ 
    result  $K(A')$ 
  then {K to USP,
    A' to arch spec units
     $K' : SP'' \rightarrow SP'$ 
     $A'' : SP''$ 
    result  $K'(A'')$ }
  then {A' to {K' to USP'}}

```

Here, “**then**” denotes composition of refinements. Details and formal semantics can be found in [46].

A second central issue concerns *behavioural refinement*. Often, a refined specification does not satisfy the initial requirements literally, but only up to some sort of behavioural equivalence. E.g. if stacks are implemented as arrays-with-pointer, then two arrays-with-pointer only differing in their “junk” entries (that is, those beyond the pointer) exhibit the same behaviour in terms of the stack operations. Hence, they correspond to the same abstract stack and should be treated as being the same for the purpose of the refinement. This can be achieved e.g. by using observational equivalences between models, which are usually induced by sets of observable sorts [59, 12].

## 9 Tools

A language will be used only if good tool support is available. The Heterogeneous Tool Set (HETS) [45] collects several tools around CASL. It provides tool support for all the layers of CASL, as well as for CASL sublanguages and extensions.

HETS consists of parsing, static analysis and proof management tools, combining various such tools for individual specification languages, thus providing a tool for heterogeneous multi-logic specification. HETS is based on a graph of logics and languages (formalized as institutions). The input language of HETS is Heterogeneous CASL (HETCASL; see [38]). HETCASL includes the structuring constructs of CASL as introduced in Sect. 6. HETCASL extends this with constructs for the translation of specifications along logic translations. The semantics of HETCASL specifications is given in terms of the so-called *Grothendieck institution* [19, 36]. This institution is basically a flattening, or disjoint union, of the logic graph.

The central device for structured theorem proving and proof management in HETS is the formalism of *development graphs*. Development graphs have been used for large industrial-scale applications [27]. The graph structure provides a direct visualization of the structure of specifications, and it also allows for managing large specifications with hundreds of sub-specifications.

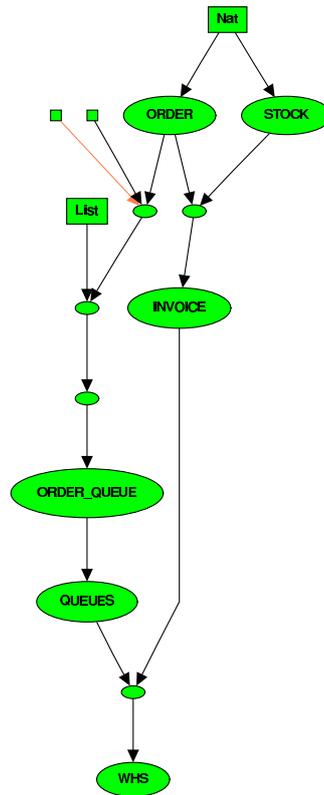


Fig. 14. Development graph for the warehouse example

A development graph (see Fig. 14 for an example graph generated by the specifications of Sect. 10) consists of a set of nodes (corresponding to whole structured specifications or parts thereof), and a set of arrows called *definition links*, indicating the dependency of each involved structured specification on its subparts. Each node is associated with a signature and some set of local axioms. The axioms of other nodes are inherited via definition links. Definition links are usually drawn as black solid arrows, denoting an import of another specification that is homogeneous (i.e. stays within the same logic). Double arrows indicate imports that are heterogeneous, i.e. changes along the arrow.

Complementary to definition links, which *define* the theories of related nodes, *theorem links* serve for *postulating* relations between different theories. Theorem links are the central data structure to represent proof obligations arising in formal developments. Theorem links can be *global* (drawn as solid arrows) or *local* (drawn as dashed arrows): a global theorem link postulates

that all axioms of the source node (including the inherited ones) hold in the target node, while a local theorem link only postulates that the local axioms of the source node hold in the target node.

The *proof calculus* for development graphs [42, 44, 40] is given by rules that allow for proving global theorem links by decomposing them into simpler (local and global) ones. Local theorem links can be proved by turning them into *local proof goals* (associated to a particular node). The latter in turn can be proved using a logic-specific calculus as given by an entailment system (see Sect. 2). Currently, the theorems provers Isabelle and SPASS have been linked to HETS, allowing for performing far more efficient reasoning compared to directly working with a calculus for basic specifications.

## 10 Case Study

This section is intended to illustrate how a system can be specified in CASL and validated/verified using the CASL tools.

As example we use the specification of a warehouse by Baumeister and Bert [7]. The warehouse is an information system that keeps track of the stock of products and orders from costumers, and it provides operations for adding, cancelling and invoicing orders, and adding products to the stock.

We both present the original specification and analyse its formal properties, at some places leading to the need of redesigning the specifications. It is quite common that not only programs, but also specification have errors and are subject to correction. A specification may be erroneous because it is ill-formed (either syntactically, or because it does not have a well-defined semantics according to the rules in Sects. 6 and 7). However, even a well-formed specification may be invalid in the sense that it does not meet the original informal specification. We will see that the calculi developed in Sects. 3.2, 6 and 7 are helpful for detecting both kinds of errors. Baumeister and Bert have revised their specifications according to the problems reported in this chapter, see [8].

First we give an overview of the specifications constituting the specification of the warehouse, and then we present the specifications in more detail, one by one. Finally we present an architectural specification that describes the modular structure of an implementation of the system.

Figure 15 gives an overview of the specifications and their extension relation. The objects of the system are products, orders, a stock and queues of pending and invoiced orders. The specifications ORDER and STOCK specify sorts, operations and predicates for orders and stocks, respectively. There is no separate specification for products, but a sort for products is declared in ORDER as well as in STOCK. The main purpose of the INVOICE specification is to specify an operation for invoicing an order in stock. The ORDER\_QUEUE and QUEUES specifications specify different kinds of queues

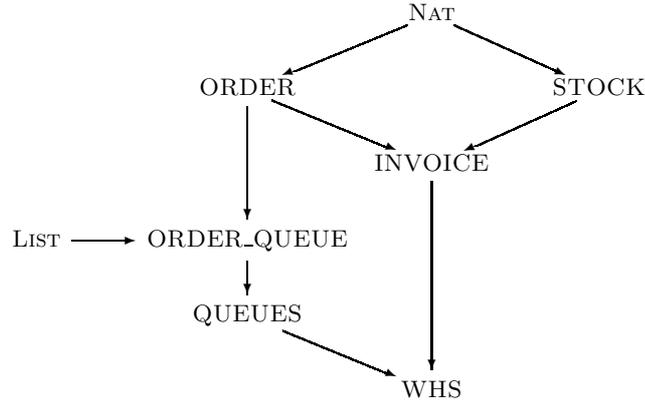


Fig. 15. Structure of the warehouse specification

of orders. The WHS specification is the top level specification in which the main operations of the system are specified.

### 10.1 Specification of Some Preliminaries

```

spec NAT = %mono
  free type Nat ::= 0 | suc(Nat)
  preds _<=_ , _>=_ : Nat × Nat
  ops   _+_ : Nat × Nat → Nat;
        _-?_ : Nat × Nat →? Nat
  ∀ m, n, r, s, t: Nat

  %% axioms concerning predicates
  • 0 <= n                                     %(leq_def1_Nat)%
  • ¬ suc(n) <= 0                             %(leq_def2_Nat)%
  • suc(m) <= suc(n) ⇔ m <= n                %(leq_def3_Nat)%
  • m >= n ⇔ n <= m                           %(geq_def_Nat)%

  %% axioms concerning operations
  • 0 + m = m                                  %(add_0_Nat)%
  • suc(n) + m = suc(n + m)                  %(add_suc_Nat)%
  • def m -? n ⇔ m >= n                       %(sub_dom_Nat)% %implied
  • m -? n = r ⇔ m = r + n                    %(sub_def_Nat)%

then %mono
  sort Pos = {p: Nat • p = 0}
  op   suc : Nat → Pos
end
  
```

NAT and LIST (the latter is shown in Fig. 3) are the usual specifications of natural numbers and lists, taken from the library of CASL basic datatypes [50]. The **free type** declaration are abbreviations for operation declarations and Peano-like axioms. For example, the **free type** declaration in the specification NAT expands to

```
spec NAT =
  sort  Nat
  ops   0 : Nat;
        suc : Nat → Nat
  ∀ X1: Nat; Y1: Nat
  • suc(X1) = suc(Y1) ⇔ X1 = Y1           %(ga_injective_suc)%
  • ¬ 0 = suc(Y1)                          %(ga_disjoint_0_suc)%
  generated {sort  Nat
             ops   0 : Nat
             suc : Nat → Nat           %(ga_generated_Nat)%}
end
```

where the **generated** construct in turn leads to a sort generation constraint  $(\{nat\}, \{0; suc\}, id)$ .

```
spec ORDER =
  NAT
  then
    sorts Order, Product
    ops  reference : Order → Product;
         ordered_qty : Order → Pos
  preds is_pending, is_invoiced : Order
  var o: Order
  • ¬ is_pending(o) ⇔ is_invoiced(o)
end
```

The ORDER specification declares a sort *Order* for orders and “observer” operations *reference* and *ordered\_qty* that for a given order give the ordered product and the ordered quantity (a positive natural number in the sort *Pos* which is a subsort of *Nat*) of this, respectively. The predicates *is\_pending* and *invoiced* test whether an order is pending or invoiced, respectively. According to the axiom an order is either pending or invoiced.

```
spec STOCK =
  NAT
  then
    sorts Stock, Product
    ops  qty : Product × Stock →? Nat;
         add : Product × Pos × Stock →? Stock;
```

```

      remove : Product × Pos × Stock →? Stock
pred  _∈_ : Product × Stock
vars  p, p' : Product; n : Pos; s : Stock
• def qty(p, s) ⇔ p ∈ s
• def add(p, n, s) ⇔ p ∈ s
• def remove(p, n, s) ⇔ p ∈ s ∧ qty(p, s) ≥ n
• qty(p, add(p, n, s)) = qty(p, s) + n if p ∈ s
• qty(p', add(p, n, s)) = qty(p', s) if p ∈ s ∧ p' ∈ s ∧ ¬ p' = p
• qty(p, remove(p, n, s)) = qty(p, s) -? n if p ∈ s ∧ qty(p, s) ≥ n
• qty(p', remove(p, n, s)) = qty(p', s) if p ∈ s ∧ p' ∈ s ∧ ¬ p' = p
end

```

The STOCK specification declares a sort *Stock* for stocks and (partial) operations *qty*, *add* and *remove* for informing about the number of items of a certain product in stock, and for adding and removing a quantity of items of a product in a stock, respectively. The predicate *is\_in* (displayed as  $\in$ ) tests whether a product is in a stock. The three first axioms specify when the partial operations are defined. The remaining axioms specify how the quantity of a product is changed by the *add* and *remove* operations.

### An Unintended Consequence

The STOCK specification has some logical consequences that are clearly revealed as not being intended when looking at the process that is being modeled.

The source of the problem is the last axiom. There, *remove*(*p,n,s*) may be undefined as its precondition  $qty(p,s) \geq n$  is not required to hold. As a consequence, we can prove that each stock contains at most one product:

$$\forall p, p' : Product; s : Stock . p \in s \wedge p' \in s \Rightarrow p = p'$$

It is certainly intended to have stocks with more than one product, hence, this consequence of the specification is not intended, showing that the specification is not in accordance with our informal understanding of the problem. This unintended consequence can be proved using the basic specification proof calculus of Sect. 3.2; however, the recognition that this reveals a discrepancy between the specification and the informal understanding of the problem is necessarily outside the scope of formal calculi.

We first prove a lemma in the specification NAT, using induction. Recall that NAT contains the sort generation constraint  $(\{nat\}, \{0; suc\}, id)$ . We apply rule **(Induction)** with

$$\Psi_{nat}(n) \equiv suc(n) \leq n \Rightarrow false.$$

This means that

$$\begin{aligned}\varphi_1 &\equiv D(0) \Rightarrow \Psi_{nat}(0) \\ \varphi_2 &\equiv D(suc(n)) \wedge \Psi_{nat}(n) \Rightarrow \Psi_{nat}(suc(n))\end{aligned}$$

We now prove  $\varphi_2$ . Assume  $D(suc(n)) \wedge \Psi_{nat}(n)$ . By ( $\wedge$ -**Elim2**),  $\Psi_{nat}(n) \equiv suc(n) \leq n \Rightarrow false$ . Assume  $suc(suc(n)) \leq suc(n)$ . By *leq\_def3\_Nat*, ( $\wedge$ -**Elim1**) and ( $\Rightarrow$ -**elim**),  $suc(n) \leq n$ . With  $\Psi_{nat}$  and ( $\Rightarrow$ -**elim**), we arrive at *false*. By ( $\Rightarrow$ -**intro**), we obtain  $suc(suc(n)) \leq suc(n) \Rightarrow false$ , which is  $\Psi_{nat}(suc(n))$ . Again by ( $\Rightarrow$ -**intro**), we obtain  $D(suc(n)) \wedge \Psi_{nat}(n) \Rightarrow \Psi_{nat}(suc(n))$ , which is just  $\varphi_2$ .

$\varphi_1$  is easy: it follows from *leq\_def2\_Nat* by (**Substitution**). By ( $\wedge$ -**Intro**), we then have  $\varphi_1 \wedge \varphi_2$ . Hence, by (**Induction**), we arrive at  $\forall n : Nat . \Psi_{nat}(n)$ , which is

$$\forall n : Nat . suc(n) \leq n \Rightarrow false \quad (1)$$

Let us now come to the proof of the unintended consequence of STOCK:

$$\forall p, p' : Product; s : Stock . p \in s \wedge p' \in s \Rightarrow p = p'$$

Using ( $\forall$ -**intro**) and ( $\Rightarrow$ -**intro**), we can reduce this to proving that  $p = p'$  follows from

$$p \in s \wedge p' \in s \quad (2)$$

We do this with rule (**Tertium non datur**). With assumption  $p = p'$ , we immediately have  $p = p'$ . It remains to show  $p = p'$  under assumption

$$p = p' \Rightarrow false \quad (3)$$

From (2) and (3) by ( $\wedge$ -**Intro**),  $p \in s \wedge p' \in s \wedge p = p' \Rightarrow false$ . With ( $\Rightarrow$ -**elim**) and the last axiom of STOCK<sup>12</sup>,  $qty(p', remove(p, n, s)) = qty(p', c)$ . By (**Function strictness**),  $D(remove(p, n, s))$ . With the third axiom of STOCK, ( $\wedge$ -**Elim1**), ( $\Rightarrow$ -**elim**) and ( $\wedge$ -**Elim2**), we arrive at

$$qty(p, s) \geq n \quad (4)$$

and by (**Predicate strictness**)

$$D(qty(p, s)) \quad (5)$$

From *geq\_defNat* in specification NAT, by ( $\wedge$ -**Elim1**),  $m \geq n \Rightarrow n \leq m$ . Rules (sum1) and (trans) of the calculus for structured specifications allow us to use this consequence also in STOCK. By (**Substitution**), we get  $D(qty(p, s)) \Rightarrow qty(p, s) \geq n \Rightarrow n \leq qty(p, s)$ . By (5) and (4) using ( $\Rightarrow$ -**elim**) twice,  $n \leq qty(p, s)$ . By (**Substitution**),

$$D(suc(qty(p, s))) \Rightarrow suc(qty(p, s)) \leq qty(p, s) \quad (6)$$

<sup>12</sup> Note that  $\psi$  if  $\varphi$  is syntactical sugar for  $\varphi \Rightarrow \psi$ , and *not*  $\varphi$  is syntactical sugar for  $\varphi \Rightarrow false$ .

From **(Totality)**, we have  $D(\text{succ}(x))$ , and with **(Substitution)**, we get  $D(\text{qty}(p, s)) \Rightarrow D(\text{succ}(\text{qty}(p, s)))$ . **( $\Rightarrow$ -elim)** with (5) gives us

$$D(\text{succ}(\text{qty}(p, s))).$$

With **( $\Rightarrow$ -elim)** and (6), we get

$$\text{succ}(\text{qty}(p, s)) \leq \text{qty}(p, s) \tag{7}$$

(1), using again the rules of the structured calculus, also is derivable in STOCK. With **( $\forall$ -Elim)** and **(Substitution)**, we get

$$D(\text{qty}(p, s)) \Rightarrow \text{succ}(\text{qty}(p, s)) \leq \text{qty}(p, s) \Rightarrow \text{false}$$

With (5) and (7) using **( $\Rightarrow$ -elim)** twice, we arrive at *false*. By **(Absurdity)**,  $p = p'$ , which is what we needed to prove.  $\square$

Of course, this proof is rather detailed and tedious. It only shows how proofs could be carried out in principle. In practice, one will use an automated or interactive theorem prover. The *Heterogeneous Tool Set* (Hets) provides an interface between CASL and the theorem prover Isabelle, which can be used to carry out the proof much more succinctly.

The unintended consequence can be avoided by adding the missing condition:

- $\text{qty}(p', \text{remove}(p, n, s)) = \text{qty}(p', s)$  if  
 $p \text{ is\_in } s \wedge p' \text{ is\_in } s \wedge \neg p' = p \wedge \text{qty}(p, s) \geq n$

## 10.2 Specification of the Warehouse System

```

spec INVOICE =
  ORDER
and
  STOCK
then
  free type
    Msg ::= success | not_pending | not_referenced | not_enough_qty
  free type OSM ::= mk(order_of:Order; stock_of:Stock; msg_of:Msg)
  pred referenced(o: Order; s: Stock)  $\Leftrightarrow$  reference(o)  $\in$  s
  pred enough_qty(o: Order; s: Stock)  $\Leftrightarrow$ 
    ordered_qty(o)  $\leq$  qty(reference(o), s)
  pred invoice_ok(o: Order; s: Stock)  $\Leftrightarrow$ 
    is_pending(o)  $\wedge$  referenced(o, s)  $\wedge$  enough_qty(o, s)
  op invoice_order : Order  $\times$  Stock  $\rightarrow$  OSM
  vars o: Order; s: Stock
  • is_invoiced(order_of(invoice_order(o, s))) if invoice_ok(o, s)
  • stock_of(invoice_order(o, s)) =

```

```

    remove(reference(o), ordered_qty(o), s) if
    invoice_ok(o, s)
    • order_of(invoice_order(o, s)) = o if ¬ invoice_ok(o, s)
    • stock_of(invoice_order(o, s)) = s if ¬ invoice_ok(o, s)
    • reference(order_of(invoice_order(o, s))) = reference(o)
    • ordered_qty(order_of(invoice_order(o, s))) = ordered_qty(o)
    • msg_of(invoice_order(o, s)) = success if invoice_ok(o, s)
    • msg_of(invoice_order(o, s)) = not_pending if ¬ is_pending(o)
    • msg_of(invoice_order(o, s)) = not_referenced if
      is_pending(o) ∧ ¬ referenced(o, s)
    • msg_of(invoice_order(o, s)) = not_enough_qty if
      is_pending(o) ∧ referenced(o, s) ∧ ¬ enough_qty(o, s)
end

```

The INVOICE specification defines a predicate *invoice\_ok* for testing the conditions for invoicing an order w.r.t. a stock: the order must be pending, the ordered product must be in stock and the ordered quantity be less than or equal to the quantity which is in stock. The definition of the predicate uses two auxiliary predicates *referenced* and *enough\_qty* also defined by this specification. (Note that the definition of predicates is written in an abbreviated syntax that expands to a predicate declaration contributing to the signature and an axiom.) The main operation of the specification is the *invoice\_order* operation for invoicing an order w.r.t. a stock. It takes an order and a stock as argument and returns updated versions of these: the state of the order is changed to invoiced and the quantity of the ordered product in stock is reduced by the ordered quantity, but only if the order is invoicable. Furthermore the operation returns a message informing whether the operation succeeded or failed because one of the conditions for invoicing failed. The effect of the operation is specified in an observational style.

```

spec ORDER_QUEUE =
  LIST [ORDER fit Elem ↦ Order] with List[Order] ↦ OQueue
then
  pred _∈_ : Order × OQueue
  vars o, o2: Order; oq: OQueue
  • ¬ o ∈ [ ]
  • o2 ∈ (o :: oq) ⇔ o2 = o ∨ o2 ∈ oq
  %% Auxiliary definitions
  ops _←_ : OQueue × Order → OQueue;
      remove : Order × OQueue → OQueue
  vars o, o2: Order; oq: OQueue
  • oq ← o = oq ++ [ o ]
  • remove(o, [ ]) = [ ]
  • remove(o, o2 :: oq) =
    o2 :: remove(o, oq) when ¬ o = o2 else remove(o, oq)
end

```

The specification `ORDER_QUEUE` defines a sort `OQueue` of queues of orders to be a list in which the elements are orders. This is done by instantiating the generic `List` specification from Fig. 3 and renaming the resulting sort `List[Order]` to `OQueue`. The predicate `is_in` tests whether an order is in a given queue. Some auxiliary operations for appending and removing an order to/from a queue are specified as well.

```

spec QUEUES =
  ORDER_QUEUE
then
  preds unicity, pqueue, iqueue : OQueue
  vars o: Order; oq: OQueue
  • unicity([ ])
  • unicity(o :: oq)  $\Leftrightarrow \neg o \in oq \wedge \text{unicity}(oq)$ 
  • pqueue(oq)  $\Leftrightarrow (\forall x: \text{Order} \bullet x \in oq \Rightarrow \text{is\_pending}(x))$ 
  • iqueue(oq)  $\Leftrightarrow (\forall x: \text{Order} \bullet x \in oq \Rightarrow \text{is\_invoiced}(x))$ 
  sorts UQueue = {oq: OQueue • unicity(oq)};
         PQueue = {uq: UQueue • pqueue(uq)};
         IQueue = {uq: UQueue • iqueue(uq)}
end

```

The `QUEUES` specification defines three subsorts of `OQueue`: `UQueue` for queues with no repetitions of orders, `PQueue` for queues only containing pending orders and `IQueue` for queues only containing invoiced orders.

```

spec WHS =
  QUEUES
and
  INVOICE
then
  free type
    GState ::= mk_gs(porders:PQueue; iorders:IQueue; the_stock:Stock)
  op   the_orders(gs: GState): OQueue = porders(gs) ++ iorders(gs)
  preds referenced(oq: OQueue; s: Stock)  $\Leftrightarrow$ 
     $\forall x: \text{Order} \bullet x \in oq \Rightarrow \text{referenced}(x, s)$ ;
    consistent(gs: GState)  $\Leftrightarrow$ 
      unicity(the_orders(gs))
       $\wedge \text{referenced}(\text{the\_orders}(\text{gs}), \text{the\_stock}(\text{gs}))$ 
  sort VGS = {gs: GState • consistent(gs)}
  pred invoiceable(pq: PQueue; s: Stock)  $\Leftrightarrow$ 
     $\exists o: \text{Order} \bullet o \in pq \wedge \text{enough\_qty}(o, s)$ 
  op   first_invoiceable : PQueue  $\times$  Stock  $\rightarrow?$  Order
  %% axioms for first_invoiceable
  vars o: Order; pq: PQueue; s: Stock
  • def first_invoiceable(pq, s)  $\Leftrightarrow \text{invoiceable}(\text{pq}, \text{s})$ 
  • first_invoiceable(o :: pq as PQueue, s) =

```

```

    o when enough_qty(o, s) else first_invoiceable(pq, s)
ops  new_order : Product × Pos × VGS → VGS;
       cancel_order : Order × VGS → VGS;
       add_qty : Product × Pos × VGS → VGS;
       deal_with_order : VGS → VGS;
       mk_order : Product × Pos × VGS → Order
%% axioms for mk_order
vars o, o1, o2: Order; p: Product; n: Pos;
       vgs: VGS; osm: OSM; s2: Stock
• is_pending(mk_order(p, n, vgs))
• ¬ mk_order(p, n, vgs) ∈ the_orders(vgs)
• reference(mk_order(p, n, vgs)) = p
• ordered_qty(mk_order(p, n, vgs)) = n

%% axioms for the warehouse operation level
• new_order(p, n, vgs) = vgs if ¬ p ∈ the_stock(vgs)
• new_order(p, n, vgs) =
  mk_gs(porders(vgs) ← mk_order(p, n, vgs) as PQueue, iorders(vgs),
  the_stock(vgs)) if
  p ∈ the_stock(vgs)
• cancel_order(o, vgs) =
  mk_gs(remove(o, porders(vgs)) as PQueue, iorders(vgs),
  the_stock(vgs))
  when o ∈ porders(vgs)
  else mk_gs(porders(vgs), remove(o, iorders(vgs)) as IQueue,
  add(reference(o), ordered_qty(o), the_stock(vgs)))
  when o ∈ iorders(vgs) else vgs
• add_qty(p, n, vgs) = vgs if ¬ p ∈ the_stock(vgs)
• add_qty(p, n, vgs) =
  mk_gs(porders(vgs), iorders(vgs), add(p, n, the_stock(vgs))) if
  p ∈ the_stock(vgs)
• deal_with_order(vgs) = vgs if
  ¬ invoiceable(porders(vgs), the_stock(vgs))
• (o1 = first_invoiceable(porders(vgs), the_stock(vgs)) ∧
  osm = invoice_order(o1, the_stock(vgs)) ∧
  o2 = order_of(osm) ∧
  s2 = stock_of(osm) ⇒
  deal_with_order(vgs) =
  mk_gs(remove(o1, porders(vgs)) as PQueue,
  iorders(vgs) ← o2 as IQueue, s2)) if
  invoiceable(porders(vgs), the_stock(vgs))
end

```

The WHS specification defines a free type of global states. The components of a global state is a queue of pending orders, a queue of invoiced orders and

a stock. A predicate *consistent* defines a desired invariant property for states, and a subtype *VGS* of consistent states is defined. A state is consistent if all orders in the queues are distinct and products referenced in the orders are products in the stock. A number of state-changing operations are declared and defined in a constructive style by the last seven axioms of the specification: *new\_order* for making a new order (i.e. adding it to the queue of pending orders, if the ordered product is in stock), *cancel\_order* for cancelling an order (i.e. removing it from the queues of orders, and if the order was invoiced also “backdating” the stock), *add\_qty* for adding a quantity of a product to the stock, if the product is in stock, and *deal\_with\_order* for dealing with an order (i.e. invoicing the first invoicable order, if any, in the queue of pending orders and moving it to the queue of invoiced orders). In order to make the specification of *new\_order* constructive, an *Order* constructor *mk\_order* is needed so this is specified as well (since *ORDER* does not provide this). In addition a number of auxiliary functions and predicates are defined.

We suggest that the *the\_orders* operation in *WHS* should return a *UQueue* instead of a *OQueue*, since this is more precise. Then the clause *unicity(the\_orders(gs))* in the definition of *consistent* could be omitted.

### 10.3 The Architectural Decomposition

Before we write an architectural specification that describes a modular structure for the implementation of *WHS*, let us point out that the simplified fragment of *CASL* architectural specifications studied in Sect. 7 forces the user to combine all the units involved in the final result expression. This often leads to quite complicated unit expressions, which can be considerably simplified by “storing” the results of subexpressions as named units within the list of unit declarations. Indeed, *CASL* provides so-called unit definitions, with a self-evident syntax and rather obvious semantics, to allow this. We use this feature in the final architectural specification below:<sup>13</sup>

```
arch spec WAREHOUSE =
  units
    NATALG: NAT;
    ORDERFUN: NAT → ORDER;
```

<sup>13</sup> In the original specification, parameterized units with two arguments are also used. Since for simplicity we cover only one-argument units in Sect. 7, at some places here we combine two units (as well as their specifications) into one.

We have also omitted in this chapter, and hence in this architectural specification as well, imports for unit specifications, used in the original specification to require the generic units *INVOICEFUN* and *WHSFUN* to work only for arguments that extend *NATALG* and, respectively, *ORDERALG* and *STOCKALG*. Since the specifications of these omitted imports are included in the specifications of unit parameters, leaving them out has no effect on the way the units can be implemented.

```

ORDERALG = ORDERFUN [NATALG];
STOCKFUN: NAT → STOCK;
STOCKALG = STOCKFUN [NATALG];
INVOICEFUN: { ORDER and STOCK } → INVOICE;
QUEUESFUN: ORDER → QUEUES;
WHSFUN : { QUEUES and INVOICE } → WHS;
result WHSFUN[QUEUESFUN[ORDERALG]
           and INVOICEFUN[ORDERALG and STOCKALG]]
end

```

The architectural specification requires a number of units that should be combined in the way explained in the result part. The unit NATALG should implement NAT. The generic units ORDERFUN should expand units implementing NAT into implementations of ORDER. Similar remarks hold for the generic units STOCKFUN and QUEUESFUN. ORDERALG and STOCKALG should be instantiations of ORDERFUN and STOCKFUN with NATALG. The generic unit INVOICEFUN should expand implementations of ORDER and STOCK into implementations of INVOICE. A similar remark holds for WHSFUN.

### A Sharing Problem

The verification semantics of architectural specifications that we have presented in Sect. 7.2 cannot be implemented in full, as it involves some true verification conditions (notably captured by the last premise in the rule for unit instantiation in Fig. 12). However, a good part of it can be implemented and provides useful support for the user. An *extended static analysis* of CASL architectural specifications can be defined along just the same lines as the verification semantics, but replacing the specifications involved with their signatures. Such an extended static analysis is implemented within the Heterogeneous Tool Set [45]. Checking the above architectural specification with this tool leads to the following error message:

```

Analyzing arch spec Warehouse
*** Error Invoice.casl:208.36-208.50, Amalgamability is not ensured:
sorts Product in OrderFun [NatAlg] and Product in StockFun [NatAlg]
might be different

```

The problem arises because *Product* is declared independently in ORDER and in STOCK. Consequently, its realisations in ORDERALG = ORDERFUN[NATALG] and STOCKALG = STOCKFUN[NATALG] may be different and therefore not amalgamable in the arguments for INVOICEFUN and WHSFUN.

Technically, this can be seen by studying the rule for unit amalgamation in Fig. 12, see also the corresponding (second) diagram in Fig. 13. Namely, elaboration of the two unit terms defining ORDERALG and STOCKALG leads

to a new sort name for *Product* in each case, and therefore two copies of it will occur in the “global signature”  $\Sigma'_G$ , one linked with the occurrence of *Product* in  $\Sigma_1$  via  $i_1; \theta'_2$ , the other with its occurrence in  $\Sigma_2$  via  $i_2; \theta'_1$  (where  $\Sigma_1$  and  $\Sigma_2$  are the signatures of ORDERALG and STOCKALG, respectively). However, the union signature  $\Sigma_1 \cup \Sigma_2$  contains only one occurrence of *Product*, and therefore for any morphism  $j: (\Sigma_1 \cup \Sigma_2) \rightarrow \Sigma'_G$ , either  $\iota_1; j \neq i_1; \theta'_2$  or  $\iota_2; j \neq i_2; \theta'_1$ . Consequently, the rule cannot be applied, and the analysis of the amalgamation expression fails.

This problem can be avoided by declaring the *Product* sort only in a separate specification PRODUCT and letting ORDER and STOCK extend PRODUCT. Then we can use the extended static semantics to show the correctness of this corrected architectural decomposition:

```

arch spec WAREHOUSE =
  units
    NATALG: NAT;
    PRODUCTALG: PRODUCT;
    ORDERFUN: { NAT and PRODUCT }  $\rightarrow$  ORDER;
    ORDERALG = ORDERFUN [NATALG and PRODUCTALG];
    STOCKFUN: { NAT and PRODUCT }  $\rightarrow$  STOCK;
    STOCKALG = STOCKFUN [NATALG and PRODUCTALG];
    INVOICEFUN: { ORDER and STOCK }  $\rightarrow$  INVOICE;
    QUEUESFUN: ORDER  $\rightarrow$  QUEUES;
    WHSFUN: { QUEUES and INVOICE }  $\rightarrow$  WHS
  result WHSFUN [QUEUESFUN [ORDERALG] and
    INVOICEFUN [ORDERALG and STOCKALG]]
end

```

Referring again to the rule for amalgamation and the corresponding diagram in Figs. 12 and 13, respectively: now the sort name for *Product* is introduced to the “global signature” only once (when PRODUCTALG is declared). It is linked with the corresponding names in both  $\Sigma_1$  and  $\Sigma_2$  (as before,  $\Sigma_1$  and  $\Sigma_2$  are the signatures of ORDERALG and STOCKALG, respectively)—consequently, the appropriate morphism  $j: (\Sigma_1 \cup \Sigma_2) \rightarrow \Sigma'_G$  exists, and the rule can be applied with no trouble.

### Inconsistent Unit Specifications

There is a problem with the parameterized units INVOICEFUN and WHSFUN above: their specifications are inconsistent. This is because both INVOICE and WHS further constrain some operation symbols occurring in the argument specification ORDER. Hence, a persistent unit function from the model class of the argument specification to that of the result specification cannot exist: those models that do not meet the further constraint cannot be mapped persistently. In WHS the problem is the *mk\_order* function and in INVOICE

it is the *invoice\_order* function. To show for example inconsistency of the specification for INVOICEFUN, notice that  $\text{INVOICE} \models \exists o_1, o_2 : \text{Order} . o_1 \neq o_2$ , because INVOICE contains a function *invoice\_order* that allows one to change the status of an order from *is\_pending* to *is\_invoiced*. On the other hand,  $\text{ORDER} \not\models \exists o_1, o_2 : \text{Order} . o_1 \neq o_2$ , because there is an ORDER-model with singleton carrier set for sort *Order*. In particular, this ORDER-model does not have an INVOICE-extension.

The deeper reason for these problems is that the specification ORDER is not detailed enough to ensure that the unit ORDERALG can be used in the way needed by the functions *mk\_order* and *invoice\_order* as specified in INVOICE and WHS. This means that the architectural specification WAREHOUSE represents an unrealistic design decision that cannot lead to an implementation.

Indeed, very loose specifications are often not sufficient in general as good specifications of the components to appear in an architectural decomposition, since often not enough information is provided to make them really usable—and so providing an architectural design may require additional details before satisfactory specifications of components are obtained.

A better design can be obtained by first refining ORDER. A way to do this is to introduce a function creating new orders. In order to be able to distinguish different orders that happen to involve the same quantity of the same product, we need to introduce labels for orders. We assume that labels are ordered and come with an order-increasing successor function, such that it is always possible to generate fresh, so far unused labels. In the following, BOOLEAN is a standard specification of the Boolean values (true, false), and RICHTOTALORDER is a specification of total orders, together with a binary maximum operation.

```

spec ORDER' = ORDER
and BOOLEAN and RICHTOTALORDER with Elem  $\mapsto$  Label
then ops   init_label : Label;
           suc : Label  $\rightarrow$  Label
           type Order ::= gen_order(reference:Product; ordered_qty:Pos;
                                   gen_pending:Boolean; label:Label)

            $\forall$  l: Label; o: Order
           • l < suc(l)
           • is_pending(o)  $\Leftrightarrow$  gen_pending(o) = True
            $\forall$  p1, p2: Product; q1, q2: Pos; b1, b2: Boolean; l1, l2: Label
           • gen_order(p1, q1, b1, l1) = gen_order(p2, q2, b2, l2)  $\Rightarrow$ 
             p1 = p2  $\wedge$  q1 = q2  $\wedge$  b1 = b2  $\wedge$  l1 = l2    %(gen_order_injective)%
end

```

INVOICE and WHS have to be refined correspondingly, such that they make use of the new function *gen\_order* generating orders:

```

spec INVOICE' = ORDER'
and INVOICE

```

```

then  $\forall o: \text{Order}; s: \text{Stock}$ 
  •  $\text{order\_of}(\text{invoice\_order}(o, s)) =$ 
     $\text{gen\_order}(\text{reference}(o), \text{ordered\_qty}(o), \text{False}, \text{label}(o))$  if
     $\text{msg\_of}(\text{invoice\_order}(o, s)) = \text{success}$ 
end

spec  $\text{WHS}' = \text{INVOICE}'$  and  $\text{WHS}$ 
then ops  $\text{max\_label} : \text{OQueue} \rightarrow \text{Label};$ 
   $\text{fresh\_label} : \text{VGS} \rightarrow \text{Label}$ 
   $\forall p: \text{Product}; n: \text{Pos}; \text{vgs}: \text{VGS}; o: \text{Order}; \text{oq}: \text{OQueue}$ 
  •  $\text{max\_label}([\ ]) = \text{init\_label}$ 
  •  $\text{max\_label}(o :: \text{oq}) = \text{max}(\text{label}(o), \text{max\_label}(\text{oq}))$ 
  •  $\text{fresh\_label}(\text{vgs}) = \text{suc}(\text{max\_label}(\text{the\_orders}(\text{vgs})))$ 
  •  $\text{mk\_order}(p, n, \text{vgs}) = \text{gen\_order}(p, n, \text{True}, \text{fresh\_label}(\text{vgs}))$ 
end

```

This finally results in a new architectural specification  $\text{WAREHOUSE}'$ :

```

arch spec  $\text{WAREHOUSE}' =$ 
  units
   $\text{NATALG}: \text{NAT};$ 
   $\text{PRODUCTALG}: \text{PRODUCT};$ 
   $\text{ORDERFUN}: \{ \text{NAT and PRODUCT} \} \rightarrow \text{ORDER}';$ 
   $\text{ORDERALG} = \text{ORDERFUN} [\text{NATALG and PRODUCTALG}];$ 
   $\text{STOCKFUN}: \{ \text{NAT and PRODUCT} \} \rightarrow \text{STOCK};$ 
   $\text{STOCKALG} = \text{STOCKFUN} [\text{NATALG and PRODUCTALG}];$ 
   $\text{INVOICEFUN}: \{ \text{ORDER}' and \text{STOCK} \} \rightarrow \text{INVOICE}';$ 
   $\text{QUEUESFUN}: \text{ORDER} \rightarrow \text{QUEUES};$ 
   $\text{WHSFUN}: \{ \text{QUEUES and INVOICE}' \} \rightarrow \text{WHS}'$ 
  result  $\text{WHSFUN} [\text{QUEUESFUN} [\text{ORDERALG}] \text{ and}$ 
     $\text{INVOICEFUN} [\text{ORDERALG and STOCKALG}]]$ 
end

```

Obviously,  $\text{ORDER}'$  and  $\text{INVOICE}'$  refine their corresponding unprimed variants. Moreover, we have the following refinement sequence:

```

refinement  $R =$ 
  WHS refined to
  WHS' refined to arch spec  $\text{WAREHOUSE}'$ 
end

```

However, note that  $\text{WAREHOUSE}'$  is *not* a refinement of  $\text{WAREHOUSE}$ : formally, this follows because  $\text{WAREHOUSE}$  is inconsistent while  $\text{WAREHOUSE}'$  is not. Indeed,  $\text{WAREHOUSE}'$  is simply a new design.

A further refinement of  $\text{WAREHOUSE}'$  would proceed for each component unit separately. For instance,

- ORDER' refines to ORDER'', where the latter replaces the sort *Label* with *Nat* (from the specification of natural numbers)<sup>14</sup>. Note though that this does *not* give extra information for use by other components unless WAREHOUSE' is changed accordingly.
- The specification {QUEUES and INVOICE'} → WHS' of the unit WHS-FUN should refine to

**arch spec**

**units**

WHSFUN' : {QUEUES and INVOICE'} → WHS'';

$F : \text{WHS}'' \rightarrow \text{WHS}'$

**result**  $\lambda Q : \{\text{QUEUES and INVOICE}'\} \bullet F[\text{WHSFUN}'[Q]]$

where WHS'' uses a more efficient method of generating fresh labels, namely, by storing the maximal label used so far as part of the state. This requires replacing the sort *VGS* with a sort involving an extra state component. The construction *F* needs to recover *VGS* from this new state sort.

## 11 Conclusion

CASL is a complex specification language providing both a complete formal semantics and a proof calculus for all of its constructs. A central property of the design of CASL is the orthogonality between on the one hand basic specifications providing means to write theories in a specific logic, and on the other hand structured and architectural specifications, which have a logic-independent semantics. This means that the logic for basic specifications can easily be changed while keeping the rest of CASL unchanged. Indeed, CASL is actually the central language in a whole family of languages. CASL concentrates on the specification of abstract data types and (first-order) functional requirements, while some (currently still prototypical) *extensions* of CASL also consider the specification of higher-order functions [43, 62] and of reactive [10, 55, 56, 57] and object-oriented [3, 26] behaviour. *Restrictions* of CASL to sublanguages [39, 37] make it possible to use specialized tool support.

Now that the design of CASL and its semantics have been completed and are laid out in a two-volume book [49, 50], the next step is to put CASL into practical use. A library of basic datatypes and several case studies have been developed in CASL [58, 1]; they show how CASL works in practice. The Heterogeneous Tool Set [45] provides tool support for all the layers of CASL, as well as for CASL extensions. Also, programming languages (formalized as particular institutions) are being integrated, leading to a framework and environment for formal software development.

<sup>14</sup> This refinement would involve a signature morphism mapping *Label* to *Nat*.

## Acknowledgements

This chapter reports results of the Common Framework Initiative (CoFI); hence we thank all the contributors to CoFI, without whom it simply would not exist. This work has been partially supported by KBN grant 7T11C 002 21 and European AGILE project IST-2001-32747, by the British-Polish Research Partnership Programme, and by the project MULTIPLE of the *Deutsche Forschungsgemeinschaft* under Grants KR 1191/5-1 and KR 1191/5-2.

## References

1. CASL case studies. Available at <http://www.pst.informatik.uni-muenchen.de/~baumeist/CoFI/case.html>.
2. S. Alagic. Institutions: integrating objects, XML and databases. *Information and Software Technology*, 44:207–216, 2002.
3. D. Ancona, M. Cerioli, and E. Zucca. Extending CASL by late binding. In C. Choppy, D. Bert, and P. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Bonas, France*, volume 1827 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
4. E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. CASL: The common algebraic specification language. *Theoretical Computer Science*, 286:153–196, 2002.
5. E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner. *Algebraic Foundations of Systems Specification*. Springer, 1999.
6. J. Barwise and J. Etchemendy. *Language, proof and logic*. CSLI publications, 2002.
7. H. Baumeister and D. Bert. Algebraic specification in CASL. In M. Frappier and H. Habrias, editors, *Software Specification Methods: An Overview Using a Case Study*, chapter 12, pages 209–224. Springer, 2001.
8. H. Baumeister and D. Bert. Algebraic specification in CASL. In M. Frappier and H. Habrias, editors, *Software Specification Methods: An Overview Using a Case Study*, chapter 15. ISTE Publishing Company, 2006.
9. H. Baumeister, M. Cerioli, A. Haxthausen, T. Mossakowski, P. Mosses, D. Sannella, and A. Tarlecki. CASL semantics. In P. Mosses, editor, *CASL Reference Manual*. [50], Part III.
10. H. Baumeister and A. Zamulin. State-based extension of CASL. In *Proceedings IFM 2000*, volume 1945 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.
11. M. Bidoit and R. Hennicker. On the integration of observability and reachability concepts. In M. Nielsen and U. Engberg, editors, *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*, volume 2303 of *Lecture Notes in Computer Science*, pages 21–36. Springer, 2002.
12. M. Bidoit, D. Sannella, and A. Tarlecki. Observational interpretation of CASL specifications. Submitted for publication, 2006.
13. T. Borzyszkowski. Generalized interpolation in CASL. *Information Processing Letters*, 76/1-2:19–24, 2000.

14. T. Borzyszkowski. Higher-order logic and theorem proving for structured specifications. In C. Choppy, D. Bert, and P. Mosses, editors, *Workshop on Algebraic Development Techniques 1999*, volume 1827 of *LNCS*, pages 401–418, 2000.
15. T. Borzyszkowski. Logical systems for structured specifications. *Theoretical Computer Science*, 286:197–245, 2002.
16. M. Cerioli, A. Haxthausen, B. Krieg-Brückner, and T. Mossakowski. Permissive subsorted partial logic in CASL. In M. Johnson, editor, *Algebraic methodology and software technology: 6th international conference, AMAST 97*, volume 1349 of *Lecture Notes in Computer Science*, pages 91–107. Springer-Verlag, 1997.
17. C. Cirstea. Institutionalising many-sorted coalgebraic modal logic. In *CMCS 2002*, *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2002.
18. CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible from <http://www.cofi.info/>.
19. R. Diaconescu. Grothendieck institutions. *Applied categorical structures*, 10:383–402, 2002.
20. T. Dimitrakos and T. Maibaum. On a generalised modularisation theorem. *Information Processing Letters*, 74(1-2):65–71, 2000.
21. J. L. Fiadeiro and J. F. Costa. Mirror, mirror in my hand: A duality between specifications and models of process behaviour. *Mathematical Structures in Computer Science*, 6(4):353–373, 1996.
22. J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39:95–146, 1992. Predecessor in: *LNCS 164*, 221–256, 1984.
23. J. A. Goguen and R. Diaconescu. Towards an algebraic semantics for the object paradigm. In *Recent Trends in Data Type Specification: Workshop on Specification of Abstract Data Types: Selected Papers*, number 785 in *LNCS*. Springer Verlag, Berlin, Germany, 1994.
24. A. Haxthausen and F. Nickl. Pushouts of order-sorted algebraic specifications. In *Proceedings of AMAST'96*, volume 1101 of *Lecture Notes in Computer Science*, pages 132–147. Springer-Verlag, 1996.
25. H. Herrlich and G. Strecker. *Category Theory*. Allyn and Bacon, Boston, 1973.
26. H. Hussmann, M. Cerioli, and H. Baumeister. From UML to CASL (static part). Technical report, 2000. Technical Report of DISI - Università di Genova, DISI-TR-00-06, Italy.
27. D. Hutter, B. Langenstein, C. Sengler, J. H. Siekmann, W. Stephan, and W. Wolpers. Verification support environment (VSE). *High Integrity Systems*, 1(6):523–530, 1996.
28. C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1990.
29. B. Klin, P. Hoffman, A. Tarlecki, L. Schröder, and T. Mossakowski. Checking amalgamability conditions for CASL architectural specifications. In *Mathematical Foundations of Computer Science*, volume 2136 of *LNCS*, pages 451–463. Springer, 2001.
30. CoFI Language Design Group, B. Krieg-Brückner and P.D. Mosses (eds.). CASL summary. In P. Mosses, editor, *CASL Reference Manual*. [50], Part I.
31. A. Lopes and J. L. Fiadeiro. Preservation and reflection in specification. In *Algebraic Methodology and Software Technology*, pages 380–394, 1997.
32. K. Meinke and J. V. Tucker, editors. *Many-sorted Logic and its Applications*. Wiley, 1993.

33. J. Meseguer. General logics. In *Logic Colloquium 87*, pages 275–329. North Holland, 1989.
34. T. Mossakowski. Colimits of order-sorted specifications. In F. Parisi Presicce, editor, *Recent trends in algebraic development techniques. Proc. 12th International Workshop*, volume 1376 of *Lecture Notes in Computer Science*, pages 316–332. Springer, 1998.
35. T. Mossakowski. Specification in an arbitrary institution with symbols. In C. Choppy, D. Bert, and P. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Bonas, France*, volume 1827 of *Lecture Notes in Computer Science*, pages 252–270. Springer-Verlag, 2000.
36. T. Mossakowski. Comorphism-based Grothendieck logics. In K. Diks and W. Rytter, editors, *Mathematical foundations of computer science*, volume 2420 of *LNCS*, pages 593–604. Springer, 2002.
37. T. Mossakowski. Relating CASL with other specification languages: the institution level. *Theoretical Computer Science*, 286:367–475, 2002.
38. T. Mossakowski. HETCASL - heterogeneous specification. Language summary, 2004.
39. T. Mossakowski. CASL sublanguages and extensions. In P. D. Mosses, editor, *CASL Reference Manual*, volume 2960 of *Lecture Notes in Computer Science*, chapter I:7, pages 61–69. Springer Verlag, London, 2004.
40. T. Mossakowski. Heterogeneous specification and the heterogeneous tool set. Habilitation thesis, University of Bremen, 2005.
41. T. Mossakowski, S. Autexier, and D. Hutter. Extending development graphs with hiding. In H. Hußmann, editor, *Fundamental Approaches to Software Engineering*, volume 2029 of *Lecture Notes in Computer Science*, pages 269–283. Springer-Verlag, 2001.
42. T. Mossakowski, S. Autexier, and D. Hutter. Development graphs – proof management for structured specifications. *Journal of Logic and Algebraic Programming*, 67(1-2):114–145, 2006.
43. T. Mossakowski, A. Haxthausen, and B. Krieg-Brückner. Subsorted partial higher-order logic as an extension of CASL. In C. Choppy, D. Bert, and P. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th International Workshop, WADT'99, Bonas, France*, volume 1827 of *Lecture Notes in Computer Science*, pages 126–145. Springer-Verlag, 2000.
44. T. Mossakowski, P. Hoffman, S. Autexier, and D. Hutter. CASL proof calculus. In P. Mosses, editor, *CASL Reference Manual*. [50], Part IV.
45. T. Mossakowski, C. Maeder, K. Lüttich, and S. Wölfl. The heterogeneous tool set. Submitted for publication. Hets is available from <http://www.tzi.de/cofi/hets>.
46. T. Mossakowski, D. Sannella, and A. Tarlecki. A simple refinement language for CASL. In J. L. Fiadeiro, editor, *WADT 2004*, volume 3423 of *Lecture Notes in Computer Science*, pages 162–185. Springer; Berlin, 2005.
47. T. Mossakowski, L. Schröder, M. Roggenbach, and H. Reichel. Algebraic-co-algebraic specification in CoCASL. *Journal of Logic and Algebraic Programming*, 67(1-2):146–197, 2006.
48. P. D. Mosses. COFI: The Common Framework Initiative for Algebraic Specification and Development. In *TAPSOFT '97, Proc. Intl. Symp. on Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 115–137. Springer-Verlag, 1997.

49. P. D. Mosses and M. Bidoit. CASL — *The Common Algebraic Specification Language: User Manual*, volume 2900 of *Lecture Notes in Computer Science*. Springer, 2004.
50. P. D. Mosses (ed.). CASL — *The Common Algebraic Specification Language: Reference Manual*, volume 2960 of *Lecture Notes in Computer Science*. Springer, 2004.
51. M. Nielsen and U. Pletat. Polymorphism in an institutional framework, 1986. Technical University of Denmark.
52. B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford Univ. Press, 1990.
53. L. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996. 2nd edition.
54. A. Popescu and G. Rosu. Behavioral extensions of institutions. In J. L. Fiadeiro, N. Harman, M. Roggenbach, and J. J. M. M. Rutten, editors, *Algebra and Coalgebra in Computer Science: First International Conference, CALCO 2005, Swansea, UK, September 3-6, 2005, Proceedings*, volume 3629 of *Lecture Notes in Computer Science*, pages 331–347. Springer, 2005.
55. G. Reggio, E. Astesiano, and C. Choppy. CASL-LTL - a CASL extension for dynamic reactive systems - summary. Technical Report of DISI - Università di Genova, DISI-TR-99-34, Italy, 2000.
56. G. Reggio and L. Repetto. CASL-CHART: a combination of statecharts and of the algebraic specification language CASL. In *Proc. AMAST 2000*, volume 1816 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.
57. M. Roggenbach. CSP-CASL — a new integration of process algebra and algebraic specification. In *Third AMAST Workshop on Algebraic Methods in Language Processing (AMiLP-3)*, TWLT. University of Twente, 2003. Long version to appear in *Theoret. Comp. Sci.*
58. M. Roggenbach, T. Mossakowski, and L. Schröder. Libraries. In P. Mosses, editor, *CASL Reference Manual*. [50], Part VI.
59. D. Sannella and A. Tarlecki. On observational equivalence and algebraic specification. *Journal of Computer and System Sciences*, 34:150–178, 1987.
60. D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation*, 76:165–210, 1988.
61. D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: Implementations revisited. *Acta Informatica*, 25:233–281, 1988.
62. L. Schröder and T. Mossakowski. HasCASL: Towards integrated specification and development of Haskell programs. In H. Kirchner and C. Reingeissen, editors, *Algebraic Methodology and Software Technology, 2002*, volume 2422 of *Lecture Notes in Computer Science*, pages 99–116. Springer-Verlag, 2002.
63. L. Schröder, T. Mossakowski, P. Hoffman, B. Klin, and A. Tarlecki. Amalgamation in the semantics of CASL. *Theoretical Computer Science*, 331(1):215–247, 2005.
64. A. Sernadas, J. F. Costa, and C. Sernadas. An institution of object behaviour. In H. Ehrig and F. Orejas, editors, *Recent Trends in Data Type Specification*, volume 785 of *Lecture Notes in Computer Science*, pages 337–350. Springer-Verlag, 1994.
65. A. Sernadas and C. Sernadas. Denotational semantics of object specification within an arbitrary temporal logic institution. Research report, Section of Com-

- puter Science, Department of Mathematics, Instituto Superior Técnico, 1049-001 Lisboa, Portugal, 1993. Presented at IS-CORE Workshop 93.
66. A. Sernadas, C. Sernadas, C. Caleiro, and T. Mossakowski. Categorical fibring of logics with terms and binding operators. In D. Gabbay and M. d. Rijke, editors, *Frontiers of Combining Systems 2*, Studies in Logic and Computation, pages 295–316. Research Studies Press, 2000.
67. A. Tarlecki. Abstract specification theory: an overview. In M. Broy and M. Pizka, editors, *Models, Algebras, and Logics of Engineering Software*, volume 191 of *NATO Science Series — Computer and System Sciences*, pages 43–79. IOS Press, 2003.

## CASL Indexes

### Symbol Index

- |   |     |  |          |
|---|-----|--|----------|
| $(\tilde{S}, \tilde{F}, \theta)$                                | 224 | $false$  | 224      |
| $\forall$ -elim   | 226 | $FO_{\Sigma}(X)$   | 224      |
| $\forall$ -intro  | 226 | $\Gamma \vdash_{\Sigma} \varphi$   | 222      |
| $\forall x : s \bullet \varphi$                                 | 224 | $\Gamma \models_{\Sigma} \varphi$  | 222      |
| $AF_{\Sigma}(X)$  | 224 | $h : M \rightarrow N$  | 223      |
| <b>arch spec</b> $Dcl^*$ <b>result</b> $T$                      | 241 | $h _{\sigma}$  | 223      |
| $B_v$   | 245 | $I$  | 221      |
| $B_v; \theta$   | 245 | INVOICE  | 258      |
| $B_{st}$  | 241 | $\mathcal{M}$  | 233      |
| $\mathcal{C}$   | 242 | $M \models \varphi$  | 225      |
| $C_{st}^{\emptyset}$  | 241 | $M \models_{\Sigma} \varphi$   | 221      |
| $C_v^{\emptyset}$   | 246 | $M_s$  | 223      |
| $\mathcal{C} \vdash T \Rightarrow UEv$                          | 242 | $M _{\Sigma}$  | 221      |
| $C_v \vdash Dcl :: C'_v$  | 247 | $M _{\sigma}$  | 221, 223 |
| $C_v = (P_v, B_v, SP_G)$  | 245 | <b>Mod</b>   | 221      |
| $C_{st} = (P_{st}, B_{st})$                                     | 241 | NAT  | 254      |
| $C_{st} \vdash T \triangleright \Sigma$                         | 241 | $\neg\varphi$  | 224      |
| $def\ t$  | 224 | $\nu^{\#}$   | 225      |
| $E$   | 242 | $\nu : X \rightarrow M$  | 225      |
| $\exists x : s \bullet \varphi$                                 | 224 | ORDER  | 255      |
| $F : \mathbf{Mod}(\Sigma_1) \rightarrow \mathbf{Mod}(\Sigma_2)$ | 242 | ORDER_QUEUE  | 259      |
| $(f_{w,s})_M$   | 223 | $P$  | 222      |
| $f : w \rightarrow? s$  | 223 | $P_v$  | 245      |
| $f : w \rightarrow s$   | 223 | $(P_v, B_v, SP_G) \vdash T :: \Sigma \xrightarrow{i} SP'_G \xleftarrow{\theta} SP_G$ | 247      |

- $(p_w)_M$  223  
 $p : w$  223  
 $p_w(t_1, \dots, t_n)$  224  
 $PCFOL^=$  222  
 $PF$  222  
 $\wedge$ -Elim1 228  
 $\wedge$ -Elim2 228  
 $\wedge$ -Intro 228  
 $\varphi \wedge \psi$  224  
 $\Rightarrow$ -elim 226  
 $\Rightarrow$ -intro 226  
 $\varphi \Rightarrow \psi$  224  
 $\varphi \vee \psi$  224  
 $P_{st}$  241  
  
 QUEUES 260  
  
 $S$  222  
**Sen** 221  
 $\Sigma$  221, 222  
 $\Sigma_1 \subseteq \Sigma_2$  221  
 $\Sigma_G$  245  
 $\widehat{\Sigma}$  229  
 $\sigma(\varphi)$  225  
 $\sigma : \Sigma \rightarrow \Sigma'$  221, 223  
 $\sigma : SP_1 \rightarrow SP_2$  233  
 $\sigma(t)$  225  
 $\sigma(\varphi)$  221  
**Sign** 221  
 $SP \vdash \varphi$  235  
 $SP_G$  245  
 $SP \models_{\Sigma} \varphi$  233  
 $SP_T$  245  
 $SP$  **then**  $SP'$  235  
 $SP_1 \approx SP_2$  233  
 $SP_1 \rightsquigarrow SP_2$  235  
**SPEC hide**  $\sigma$  233  
**SPEC with**  $\sigma$  233  
**SPEC<sub>1</sub> then free {SPEC<sub>2</sub>}** 233  
**SPEC<sub>1</sub> and SPEC<sub>2</sub>** 233  
 STOCK 255  
  
 $T_{\Sigma}(X)$  224  
 $T_1$  **and**  $T_2$  241  
 $t \stackrel{e}{=} t'$  224  
  
 $t \stackrel{s}{=} t'$  224  
 $TF$  222  
 $true$  224  
  
 $U : SP$  241  
 $U : SP_1 \xrightarrow{\tau} SP_2$  241  
 $U[T \text{ fit } \sigma]$  241  
 $UEv$  242  
  
 $v : SP_1 \rightarrow SP_2$  239  
  
 WHS 260  
  
 $X$  224  
 $X_s$  224  
  
 $\vdash$  222  
 $\vdash ASP :: SP$  246  
 $\vdash SP \Rightarrow \mathcal{M}$  233  
 $\vdash SP \triangleright \Sigma$  233  
 $\vdash UDD^* \Rightarrow \mathcal{C}$  242  
 $\vdash UDD^* \triangleright C_{st}$  241  
 $\models$  221  
  
**Concept Index**  
  
 Absurdity (proof rule) 226  
 Algebraic specification 219  
 amalgamation 236, 244  
 annotations 232  
 architectural design 265  
 architectural specification 239  
     internal correctness 244  
     proof rules 246  
 argument specification 238  
 associativity annotation 232  
 atomic formula 224  
  
 Basic (proof rule) 235  
 basic (proof rule) 235  
 basic datatypes 267  
 basic specification 222  
 behavioural refinement 251  
 borrowing of proofs 231  
  
 carrier set 223

- CASL language 231
- cell calculus 249
- CoFI 220
- Common Framework Initiative 220
- compatible models 237
- completeness 222
  - of basic calculus 228
  - of structured calculus 237
- Congruence (proof rule) 226
- conjunction
  - institution with 237
- conservative extension 235
  - in CASL 237
- consistent specification 233, 238
- constraint
  - sort generation 224
- constructor refinement 250
- correctness
  - of architectural specification 244
- CR (proof rule) 235
- Craig interpolation property 236
  
- datatype 232
- definedness assertion 224
- definition link 252
- derivation 227
- Derive (proof rule) 235
- derive (proof rule) 235
- design specification 250
- development graph 239, 251
- diagram semantics 249
  
- Eigenvariable conditions 227
- embedding function 229
- enriched CASL 249
- entailment system 222
- exhaustive signature unions 241
- existential equation 224
- extended static analysis 263
- extensions of CASL 267
  
- first-order formula 224
- first-order logic 222
- fits
  - unit context fits static context 243
  - unit context fits verification context 247
- fitting morphism 238
- formula
  - atomic 224
  - first-order 224
- free datatype 232
- Function Strictness (proof rule) 226
  
- generated datatype 232
- generic specification 238
- global signature 245
- global specification 245
  
- HetCASL 251
- Heterogeneous Tool Set 251
- Hets 251
- homomorphism 223
  - subsorted 230
  
- implementation 250
- implication
  - institution with 237
- import (of a specification) 238
- incompleteness
  - of basic calculus 228
- inconsistent specification 264
- Induction (proof rule) 226
- institution 221
  - with conjunction 237
  - with implication 237
  - with symbols 234
  - with unions 221
- Isabelle theorem prover 253
  
- local environment 235
- logic 222
- logical consequence 222
  
- many-sorted
  - model 223
  - partial first-order logic 222
  - signature 222
- membership predicate 230

- mixfix syntax 232
- model 221
  - many-sorted 223
  - subsorted 230
- model semantics 232
- model-oriented specification 220
- models
  - compatible 237
- named specification 238
- oracle for conservative extensions 235
- overloading relation 229
- parameterized specification 238
- parameterized unit 239
- persistent unit function 239
- precedence annotation 232
- Predicate Strictness (proof rule) 226
- program 219
- projection function 230
- proof calculus
  - for architectural specifications 246
  - for basic specifications 226
  - for development graphs 253
  - for free specifications 238
  - for refinement 235
  - for structured specifications 235
  - for subsorted specifications 231
- property-oriented specification 220
- reduct 221, 223
  - subsorted 230
- refinement 250
- Reflexivity (proof rule) 226
- requirement specification 250
- restrictions of CASL 267
- satisfaction 225
  - relation 221
  - subsorted 231
- satisfaction condition 221
- semantical entailment 222
- sentence 221, 224
  - subsorted 230
- sharing (among units) 244, 263
- signature morphism 223
  - subsorted 229
- sort generation constraint 224
- Sortgen-intro (proof rule) 226
- soundness 222
  - of basic calculus 228
  - of structured calculus 235
- SPASS theorem prover 253
- specification
  - architectural 239
  - basic 222
  - consistent 233, 238
  - design 250
  - generic 238
  - inconsistent 264
  - model-oriented 220
  - named 238
  - parameterized 238
  - property-oriented 220
  - requirement 250
  - structured 233
  - subsorted 229
- specification fragment 235
- specification morphism 233
- static context 241
- static semantics 232
- strong equation 224
- structured specification 233
- subsignature 221
- subsort 229
- subsort relation 229
- subsorted
  - homomorphism 230
  - model 230
  - reduct 230
  - satisfaction 231
  - sentence 230
  - signature 229
  - signature morphism 229
  - specification 229
- Substitution (proof rule) 226
- Sum (proof rule) 235

- sum1 (proof rule) 235
- sum2 (proof rule) 235
- symbol map 234
  
- term 224
- term evaluation 225
- Tertium non datur (proof rule) 226
- theorem link 252
- Totality (proof rule) 226
- Trans (proof rule) 235
- trans (proof rule) 235
- Trans-equiv (proof rule) 235
- Trans1 (proof rule) 235
- Trans2 (proof rule) 235
  
- unions
  - institution with 221
- unit 239
  - context 242
  - environment 242
  - evaluator 242
  - expression 240
  
- variable system 224
- variable valuation 225
- verification context 245
- view 239
  
- weakly amalgamable 236
- witnessed (unit environment is witnessed by unit) 247

