Computing and Informatics, Vol. 20, 2001, **??**–**??**, V 2003-Jun-20

CASL — THE COMMON ALGEBRAIC SPECIFICATION LANGUAGE: SEMANTICS AND PROOF THEORY

Till Mossakowski

Department of Computer Science, University of Bremen, Germany
e-mail: `till@tzi.de`

Anne Haxthausen

Informatics and Mathematical Modelling, Techn. University of Denmark, Lyngby
e-mail: `ah@imm.dtu.dk`

Donald Sannella

LFCS, School of Informatics, University of Edinburgh, Edinburgh, UK
e-mail: `dts@inf.ed.ac.uk`

Andrzej Tarlecki

Institute of Informatics, Warsaw University and Institute of Computer Science, PAS, Warsaw, Poland
e-mail: `tarlecki@mimuw.edu.pl`

Manuscript received ...
Communicated by...

Abstract. 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 structure of implementations).

We describe an simplified version of the CASL syntax, semantics and proof calculus at 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.

Keywords: Algebraic specification, formal software development, logic, calculi, institutions

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 [18] 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. [31]), 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 [3] is an expressive specification language that has been designed by CoFI, the international *Common Framework Initiative for algebraic specification and development* [29, 11], 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 paper gives an overview of the semantic concepts and proof calculi underlying CASL. Section 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 providing a proof calculus for reasoning within such unstructured specifications. The institution underlying CASL, together with its proof calculus, is presented in Sections 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*, expressing how more complex specifications are built from simpler ones (Section 6). The semantics and proof calculus is given in a way that is parameterized over the particular institution and proof calculus for basic specifications (as given at the basic specification layer). 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 structure of the *implementation*, with the possibility of enforcing a separate development of composable, reusable implementation units (Section 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.

Due to space limitations, this paper only covers a simplified version of CASL, and mainly introduces semantic concepts, while language constructs are only briefly treated in Section 5. A full account of CASL (also covering libraries of specifications) will appear in [30], see also [12, 13, 3, 25].

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 [14].

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

- a category $\mathbf{Sign}$ of *signatures*,
- a functor $\mathbf{Sen}\colon \mathbf{Sign} \to \mathbf{Set}$ giving, for each signature Σ, a set of *sentences* $\mathbf{Sen}(\Sigma)$, and for each signature morphism $\sigma\colon \Sigma \to \Sigma'$, a *sentence translation map* $\mathbf{Sen}(\sigma)\colon \mathbf{Sen}(\Sigma) \to \mathbf{Sen}(\Sigma')$, where $\mathbf{Sen}(\sigma)(\varphi)$ is often written $\sigma(\varphi)$,
- a functor $\mathbf{Mod}\colon \mathbf{Sign}^{op} \to \mathcal{CAT}$ [1] giving, for each signature Σ, a category of *models* $\mathbf{Mod}(\Sigma)$, and for each signature morphism $\sigma\colon \Sigma \to \Sigma'$, a *reduct functor* $\mathbf{Mod}(\sigma)\colon \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$, where $\mathbf{Mod}(\sigma)(M')$ is often written $M'|_\sigma$,
- a satisfaction relation $\models_\Sigma \subseteq |\mathbf{Mod}(\Sigma)| \times \mathbf{Sen}(\Sigma)$ for each $\Sigma \in \mathbf{Sign}$,

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

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

[1] 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 [16].

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" $\iota_1 \colon \Sigma_1 \to \Sigma_1 \cup \Sigma_2$ and $\iota_2 \colon \Sigma_2 \to \Sigma_1 \cup \Sigma_2$. We write $M|_{\Sigma_i}$ for $M|_{\iota_i \colon \Sigma_i \to \Sigma_1 \cup \Sigma_2}$ $(i = 1, 2)$ whenever ι_i is clear from the context.

Within an arbitrary but fixed institution, we can easily define the usual notion of *logical consequence* or *semantical entailment*. Given a set of Σ-sentences Γ and a Σ-sentence φ, we say that φ *follows from* Γ, written $\Gamma \models_\Sigma \varphi$, iff for all Σ-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 [21] 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:

1. *reflexivity:* for any $\varphi \in \mathbf{Sen}(\Sigma)$, $\{\varphi\} \vdash_\Sigma \varphi$,
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 $\sigma \colon \Sigma \to \Sigma'$ 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 [13, 10]. In this section, we introduce the institution of many-sorted partial first-order logic with sort generation constraints and equality, $PCFOL^=$. In Section 4, subsorting is added.

3.1.1 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 $f : w \to s \in TF$ for $f \in TF_{w,s}$ (with $f : s$ for empty w), $f : w \to ?\ s \in PF$ for $f \in PF_{w,s}$ (with $f : \to ?\ s$ for empty w) and $p : \P w \in P$ 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 Section 5), such qualifications may be omitted when they are unambiguously determined by the context.

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

3.1.2 Models

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

A *many-sorted* Σ*-model* M consists of a non-empty *carrier set* M_s for each sort $s \in S$, a total function $(f_{w,s})_M : M_w \to M_s$ for each total function symbol $f : w \to s \in TF$, a partial function $(f_{w,s})_M : M_w \rightharpoonup M_s$ for each partial function symbol $f : w \to ?\ s \in PF$, and a predicate $(p_w)_M \subseteq M_w$ for each predicate symbol $p : \P w \in P$. Requiring carriers to be non-empty simplifies deduction and makes it unproblematic to regard axioms (see Section 3.1.3) as implicitly universally quantified.

A *many-sorted* Σ*-homomorphism* $h : M \to N$ 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 $\mathbf{Mod}(\Sigma)$.

Concerning *reducts*, if $\sigma : \Sigma \to \Sigma'$ is a signature morphism and M' is a Σ'-model, then $M'|_\sigma$ is a Σ-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 Σ'-homomorphism $h' : M' \to N'$: its reduct $h'|_\sigma : M'|_\sigma \to N'|_\sigma$ is the Σ-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 **Mod** is a functor.

3.1.3 Sentences

Let $\Sigma = (S, TF, PF, P)$. A *variable system over* Σ 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* Σ*-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 Σ-model by defining $(f_{w,s})_{T_\Sigma(X)}$ to be the term-formation operations for $f : w \to s \in TF$ and $f : w \to ?\ s \in PF$, and $(p_w)_{T_\Sigma(X)} = \emptyset$ for $p : \P w \in P$.

An atomic Σ-formula is either: an application $p_w(t_1, \ldots, t_n)$ 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* Σ*-formulas* with variables in X. The set $FO_\Sigma(X)$ of *many-sorted first-order* Σ*-formulas* with variables in X is then defined by adding a formula F (false) and closing under conjunction $\varphi \wedge \psi$, implication $\varphi \Rightarrow \psi$ and universal quantification $\forall x : s \bullet \varphi$. We use the usual abbreviations: $\neg\varphi$ for $\varphi \Rightarrow F$, $\varphi \vee \psi$ for $\neg(\neg\varphi \wedge \neg\psi)$, T for $\neg F$ 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 Σ is a triple $(\tilde{S}, \tilde{F}, \theta)$, where $\theta\colon \overline{\Sigma} \to \Sigma$, $\overline{\Sigma} = (\overline{S}, \overline{TF}, \overline{PF}, \overline{P})$, $\tilde{S} \subseteq \overline{S}$ and $\tilde{F} \subseteq \overline{TF} \cup \overline{PF}$.

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

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

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

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

$$\zeta^{\#}_{\sigma,X}\colon T_\Sigma(X) \to T_{\Sigma'}(\sigma(X))|_\sigma$$

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

constraint $(\widetilde{S}, \widetilde{F}, \theta;\sigma)$. It is easy to see that sentence translation preserves identities and composition, so sentence translation is functorial.

3.1.4 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 $\nu\colon X \to M$ for X in M, *term evaluation* $\nu^{\#} : T_\Sigma(X) \rightharpoonup M$ is defined in the obvious way, with $t \in dom(\nu^{\#})$ 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 φ in a model M is always defined, and it is either true or false: that is, we have a two-valued logic. The application $p_w(t_1, \ldots, t_n)$ of a predicate symbol to a sequence of argument terms is satisfied with respect to a valuation $\nu\colon X \to M$ iff the values of all of $t_1, \ldots, t_n$ 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 notions of equation coincide for defined terms. Satisfaction of other formulae is defined in the obvious way. A formula φ is satisfied in a model M, written $M \models \varphi$, iff it is satisfied with respect to all variable valuations into M.

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

For a sort generation constraint $(\widetilde{S}, \widetilde{F}, \theta)$ we can assume without loss of generality that all the result sorts of function symbols in $\widetilde{F}$ occur in $\widetilde{S}$. If not, we can just omit from $\widetilde{F}$ those function symbols not satisfying this requirement, without affecting satisfaction of the sort generation constraint: in the $\overline{\Sigma}$-term t witnessing the satisfaction of the constraint, any application of a function symbol with result sort outside $\widetilde{S}$ 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 [25].

3.2 Proof calculus

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

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,

(Absurdity) $$\frac{\mathit{false}}{\varphi}$$ **(Tertium non datur)** $$\frac{\begin{array}{cc}[\varphi] & [\varphi \Rightarrow \mathit{false}] \\ \vdots & \vdots \\ \psi & \psi\end{array}}{\psi}$$

($\Rightarrow$-intro) $$\frac{\begin{array}{c}[\varphi] \\ \vdots \\ \psi\end{array}}{\varphi \Rightarrow \psi}$$ **($\Rightarrow$-elim)** $$\frac{\begin{array}{c}\varphi \\ \varphi \Rightarrow \psi\end{array}}{\psi}$$ **($\forall$-elim)** $$\frac{\forall x : s.\varphi}{\varphi}$$

($\forall$-intro) $$\frac{\varphi}{\forall x : s.\varphi}$$ where x_s occurs freely only in local assumptions

(Reflexivity) $$\frac{}{x_s \stackrel{e}{=} x_s}$$ if x_s is a variable

(Congruence) $$\frac{\varphi}{(\bigwedge_{x_s \in FV(\varphi)} x_s \stackrel{e}{=} \nu(x_s)) \Rightarrow \varphi[\nu]}$$ if $\varphi[\nu]$ defined

(Substitution) $$\frac{\varphi}{(\bigwedge_{x_s \in FV(\varphi)} D(\nu(x_s))) \Rightarrow \varphi[\nu]}$$
if $\varphi[\nu]$ defined and $FV(\varphi)$ occur freely only in local assumptions

(Totality) $$\frac{}{D(f_{w,s}\langle x_{s_1}, \ldots, x_{s_n}\rangle)}$$ if $w = s_1 \ldots s_n, f \in TF_{w,s}$

(Function Strictness) $$\frac{t_1 \stackrel{e}{=} t_2}{D(t)}$$ t some subterm of t_1 or t_2

(Predicate Strictness) $$\frac{p_w\langle t_1, \ldots, t_n\rangle}{D(t_i)}$$ $i \in \{1, \ldots, n\}$

(Induction) $$\frac{\begin{array}{c}(S, F, \theta \colon \bar{\Sigma} \to \Sigma) \\ \varphi_1 \wedge \cdots \wedge \varphi_k\end{array}}{\bigwedge_{s \in S} \forall x : \theta(s) \bullet \Psi_s(x)}$$
$F = \{f_1 \colon s^1_1 \ldots s^1_{m_1} \to s^1; \ldots; f_k \colon s^k_1 \ldots s^k_{m_k} \to s^k\}$,
Ψ_{s_j} is a formula with one free variable x of sort $\theta(s_j), j = 1, \ldots, k$,
$$\varphi_j = \forall x_1 : \theta(s^j_1), \ldots, x_{m_j} : \theta(s^j_{m_j}) \bullet \Big(D(\theta(f_j)(x_1, \ldots, x_{m_j})) \wedge \bigwedge_{i=1,\ldots,m_j;\ s^j_i \in S} \Psi_{s^j_i}(x_i)\Big) \Rightarrow \Psi_{s_j}\Big(\theta(f_j)(x_1, \ldots, x_{m_j})\Big)$$

(Sortgen-intro) $$\frac{\varphi_1 \wedge \cdots \wedge \varphi_k \Rightarrow \bigwedge_{s \in S} \forall x : \theta(s) \bullet p_s(x)}{(S, F, \theta \colon \bar{\Sigma} \to \Sigma)}$$
$F = \{f_1 \colon s^1_1 \ldots s^1_{m_1} \to s^1; \ldots; f_k \colon s^k_1 \ldots s^k_{m_k} \to s^k\}$,
for $s \in S$, the predicates $p_s : \theta(s)$ occur only in local assumptions,
and for $j = 1, \ldots, k$,
$$\varphi_j = \forall x_1 : \theta(s^j_1), \ldots, x_{m_j} : \theta(s^j_{m_j}) \bullet \Big(D(\theta(f_j)(x_1, \ldots, x_{m_j})) \wedge \bigwedge_{i=1,\ldots,m_j;\ s^j_i \in S} p_{s^j_i}(x_i)\Big) \Rightarrow p_{s_j}\Big(\theta(f_j)(x_1, \ldots, x_{m_j})\Big)$$

Fig. 1. Deduction rules for CASL basic specifications

$D(t)$, is just an abbreviation for the existential equality $t \stackrel{e}{=} t$.) Totality, Function Strictness and Predicate Strictness have self-explanatory names; they allow to infer definedness statements. Finally, the last two rules deal with sort generation constraints. If these are seen as second-order universally quantified formulas, Induction corresponds to second-order $\forall$-Elim, and Sortgen-Intro corresponds to second-order $\forall$-Intro. The φ_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 parallel induction running simultaneously over several sorts).

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

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

If Φ and φ consist of Σ-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.

The following theorem is proved in [30]:

Theorem 1. The above proof calculus yields an entailment system. Equipped with this entailment system, the CASL institution $SubPCFOL^=$ 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 (up to isomorphism). By Gödel's incompleteness theorem, there cannot be a recursively axiomatized complete calculus for such systems.

Theorem 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 [25].

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 [15, 22]), 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 Section 3 has to be modified slightly. First, in Section 4.1.1, the category of 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, in Sections 4.1.2–4.1.4, 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 ($SubPCFOL^=$).

4.1.1 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 $f : w_1 \to s_1$ (or $f : w_1 \to ?\ s_1$) and $f : w_2 \to s_2$ (or $f : w_2 \to ?\ s_2$) 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 $p : \P w_1$ and $p : \P w_2$ 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* $\sigma : \Sigma \to \Sigma'$ 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 $\hat{\Sigma}$, which is the extension of the underlying many-sorted signature (S, TF, PF, P) with

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

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

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

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

4.1.2 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 Σ are ordinary many-sorted models for $\widehat{\Sigma}$ satisfying a set $Ax(\Sigma)$ of sentences 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 Σ*-homomorphisms* are ordinary many-sorted $\widehat{\Sigma}$-homomorphisms.

Hence, the category of subsorted Σ-models $\mathbf{SubMod}(\Sigma)$ is a full subcategory of $\mathbf{Mod}(\widehat{\Sigma})$.

The *reduct* of Σ'-models and Σ'-homomorphisms along a subsorted signature morphism σ from Σ into Σ' is the many-sorted reduct along the signature morphism $\widehat{\sigma}$.

4.1.3 Sentences

For a subsorted signature Σ, 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 σ is the ordinary many-sorted translation along $\widehat{\sigma}$.

4.1.4 Satisfaction

Since subsorted Σ-models and Σ-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 borrowed from the many-sorted case. To prove that a Σ-sentence φ is a Σ-consequence of a set of assumptions Φ, one just has to prove that φ is a $\hat{\Sigma}$-consequence of Φ and $Ax(\Sigma)$, i.e.

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

if and only if

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

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 paper, 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 [12] and the CASL semantics [13].

The CASL language provides constructs for declaring sorts, subsorts, operations[2] 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.

```
%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. 2. Specification of lists over an arbitrary element sort in CASL.

[2] At the level of constructs, functions are called operations.

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.

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.

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.

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. 2. The translation of CASL constructs to the underlying mathematical concepts is formally defined in the CASL semantics [13], 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 Section 2. Indeed, many different logics, including first-order, higher-order, polymorphic, modal, temporal, behavioural, process, and object-oriented logics have been shown to be institutions.

$$
\begin{array}{lll}
\texttt{SPEC} & ::= & \texttt{BASIC-SPEC} \\
& \mid & \texttt{SPEC}_1 \ \textbf{and} \ \texttt{SPEC}_2 \\
& \mid & \texttt{SPEC} \ \textbf{with} \ \sigma \\
& \mid & \texttt{SPEC} \ \textbf{hide} \ \sigma \\
& \mid & \texttt{SPEC}_1 \ \textbf{then free} \ \{ \texttt{SPEC}_2 \}
\end{array}
$$

Fig. 3. Simplified syntax of CASL structured specifications

$$
\frac{\vdash \texttt{BASIC-SPEC} \rhd \langle \Sigma, \Gamma \rangle}{\vdash \texttt{BASIC-SPEC} \rhd \Sigma}
\qquad
\frac{\begin{array}{c}\vdash \texttt{BASIC-SPEC} \rhd \langle \Sigma, \Gamma \rangle \\ \mathcal{M} = \{M \in Mod[\Sigma] \mid M \models \Gamma\}\end{array}}{\vdash \texttt{BASIC-SPEC} \Rightarrow \mathcal{M}}
$$

$$
\frac{\begin{array}{c}\vdash SP_1 \rhd \Sigma_1 \\ \vdash SP_2 \rhd \Sigma_2 \\ \Sigma' = \Sigma_1 \cup \Sigma_2 \text{ is defined}\end{array}}{\vdash SP_1 \ \textbf{and} \ SP_2 \rhd \Sigma'}
\qquad
\frac{\begin{array}{c}\vdash SP_1 \rhd \Sigma_1 \quad \vdash SP_2 \rhd \Sigma_2 \\ \Sigma' = \Sigma_1 \cup \Sigma_2 \text{ is defined} \\ \vdash SP_1 \Rightarrow \mathcal{M}_1 \quad \vdash SP_2 \Rightarrow \mathcal{M}_2 \\ \mathcal{M} = \{M \in Mod[\Sigma'] \mid M|_{\Sigma_i} \in \mathcal{M}_i,\ i = 1, 2\}\end{array}}{\vdash SP_1 \ \textbf{and} \ SP_2 \Rightarrow \mathcal{M}}
$$

$$
\frac{\vdash SP \rhd \Sigma}{\vdash SP \ \textbf{with} \ \sigma\colon \Sigma \to \Sigma' \rhd \Sigma'}
\qquad
\frac{\begin{array}{c}\vdash SP \rhd \Sigma \quad \vdash SP \Rightarrow \mathcal{M} \\ \mathcal{M}' = \{M \in Mod[\Sigma'] \mid M|_\sigma \in \mathcal{M}\}\end{array}}{\vdash SP \ \textbf{with} \ \sigma\colon \Sigma \to \Sigma' \Rightarrow \mathcal{M}'}
$$

$$
\frac{\vdash SP \rhd \Sigma'}{\vdash SP \ \textbf{hide} \ \sigma\colon \Sigma \to \Sigma' \rhd \Sigma}
\qquad
\frac{\begin{array}{c}\vdash SP \rhd \Sigma' \quad \vdash SP \Rightarrow \mathcal{M} \\ \mathcal{M}' = \{M|_\sigma \mid M \in \mathcal{M}\}\end{array}}{\vdash SP \ \textbf{hide} \ \sigma\colon \Sigma \to \Sigma' \Rightarrow \mathcal{M}'}
$$

$$
\frac{\begin{array}{c}\vdash SP_1 \rhd \Sigma_1 \quad \vdash SP_2 \rhd \Sigma_2 \\ \Sigma_1 \subseteq \Sigma_2\end{array}}{\vdash SP_1 \ \textbf{then free} \ \{ SP_2 \} \rhd \Sigma_2}
$$

$$
\frac{\begin{array}{c}\vdash SP_1 \rhd \Sigma_1 \quad \vdash SP_2 \rhd \Sigma_2 \\ \iota\colon \Sigma_1 \to \Sigma_2 \text{ is the inclusion} \\ \vdash SP_1 \Rightarrow \mathcal{M}_1 \quad \vdash SP_2 \Rightarrow \mathcal{M}_2 \\ \mathcal{M}' = \{M \mid M \text{ is } \mathbf{Mod}(\iota)\text{-free over } M\big|_\iota \text{ in } \mathcal{M}_2\}\end{array}}{\vdash SP_1 \ \textbf{then free} \ \{ SP_2 \} \Rightarrow \mathcal{M}'}
$$

M being $\mathbf{Mod}(\iota)$-free over $M\big|_\iota$ in $\mathcal{M}_2$ means that for each model $M' \in \mathcal{M}_2$ and model morphism $h\colon M\big|_\iota \to M'\big|_\iota$, there exists a unique model morphism $h^\#\colon M \to M'$ with $h^\#\big|_\iota = h$.

Fig. 4. Semantics of 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. 3. The syntax of basic specifications `BASIC-SPEC` (as well as that of signature morphisms σ) is left unexplained, since it is provided together with the institution.

Fig. 4 shows the semantics of structured specifications [37, 13]. The static semantics is shown on the left of the figure, using judgements of the form $\vdash$ *phrase* $\rhd$ *result*. The model semantics is shown on the right, using judgements of the form $\vdash$ *phrase* $\Rightarrow$ *result*.

Using the model semantics, we can define semantical entailment as follows: a well-formed Σ-specification SP entails a Σ-sentence φ, written $SP \models_\Sigma \varphi$, if φ is satisfied in all SP-models. Moreover, we also have a simple notion of refinement between specifications: SP_1 refines to SP_2, written $SP_1 \leadsto SP_2$, if every SP_2-model is also an SP_1-model. Given a Σ_1-specification SP_1 and a Σ_2-specification SP_2, a *specification morphism* $\sigma\colon SP_1 \to SP_2$ is a signature morphism $\sigma\colon \Sigma_1 \to \Sigma_2$ such that for each SP_2-model M, $M|_\sigma$ is an SP_1-model. Note that $\sigma\colon SP_1 \to SP_2$ is a specification morphism iff $SP_1 \leadsto SP_2$ **hide** σ.

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. Details can be found in [13, 24].

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. Again, details can be found in [13].

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 [9] for proving entailments of the form $SP \vdash \varphi$, where SP is a structured specification and φ is a formula, see Fig. 5. Fig. 6 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 $\sigma\colon SP_1 \rightarrow SP_2$ is conservative iff each SP_1-model is the σ-reduct of some SP_2-model.[3]

$$(CR)\ \frac{\{SP \vdash \varphi_i\}_{i\in I} \quad \{\varphi_i\}_{i\in I} \vdash \varphi}{SP \vdash \varphi} \qquad (basic)\ \frac{\varphi \in \Gamma}{\langle \Sigma, \Gamma\rangle \vdash \varphi}$$

$$(sum1)\ \frac{SP_1 \vdash \varphi}{SP_1 \textbf{ and } SP_2 \vdash \iota_1(\varphi)} \qquad (sum2)\ \frac{SP_2 \vdash \varphi}{SP_1 \textbf{ and } SP_2 \vdash \iota_2(\varphi)}$$

$$(trans)\ \frac{SP \vdash \varphi}{SP \textbf{ with } \sigma \vdash \sigma(\varphi)} \qquad (derive)\ \frac{SP \vdash \sigma(\varphi)}{SP \textbf{ hide } \sigma \vdash \varphi}$$

Fig. 5. Proof calculus for entailment in structured specifications

$$(Basic)\ \frac{SP \vdash \Gamma}{\langle \Sigma, \Gamma\rangle \rightsquigarrow SP} \qquad (Sum)\ \frac{SP_1 \textbf{ with } \iota_1 \rightsquigarrow SP \quad SP_2 \textbf{ with } \iota_2 \rightsquigarrow SP}{SP_1 \textbf{ and } SP_2 \rightsquigarrow SP}$$

$$(Trans_1)\ \frac{SP \rightsquigarrow SP' \textbf{ with } \theta \quad \theta = \sigma^{-1}}{SP \textbf{ with } \sigma \rightsquigarrow SP'} \qquad (Trans_2)\ \frac{SP \rightsquigarrow SP' \textbf{ hide } \sigma}{SP \textbf{ with } \sigma \rightsquigarrow SP'}$$

$$(Derive)\ \frac{SP \rightsquigarrow SP''}{SP \textbf{ hide } \sigma \rightsquigarrow SP'} \quad \text{if } \sigma\colon SP' \rightarrow SP'' \text{ is a conservative extension}$$

$$(Trans\text{-}equiv)\ \frac{(SP \textbf{ with } \sigma) \textbf{ with } \theta \rightsquigarrow SP'}{SP \textbf{ with } \sigma;\theta \rightsquigarrow SP'}$$

Fig. 6. Proof calculus for refinement of structured specifications

Theorem 3 (Soundness [9])**.** 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 $SP_1 \ \approx\!\!\!\!\!\rightsquigarrow\ SP_2$.

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

[3] 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.

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

$$\begin{tikzcd} \Sigma \arrow[r, "\sigma_1"] \arrow[d, "\sigma_2"'] & \Sigma_1 \arrow[d, "\theta_2"'] \\ \Sigma_2 \arrow[r, "\theta_1"] & \Sigma' \end{tikzcd}$$

any Σ_1-sentence φ_1 and any Σ_2-sentence φ_2 with

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

there exists a Σ-sentence φ (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. (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

$$\begin{tikzcd} \Sigma \arrow[r] \arrow[d] & \Sigma_1 \arrow[d] \\ \Sigma_2 \arrow[r] & \Sigma_R \end{tikzcd}$$

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

$$\begin{tikzcd} \mathbf{Mod}(\Sigma) & \mathbf{Mod}(\Sigma_1) \arrow[l] \\ \mathbf{Mod}(\Sigma_2) \arrow[u] & \mathbf{Mod}(\Sigma_R) \arrow[l] \arrow[u] \end{tikzcd}$$

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 Σ-model can be *amalgamated* to a unique (or weakly amalgamated to a not necessarily unique) Σ_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 Σ-sentences φ_1 and φ_2, there is a Σ-sentence φ that holds in a model iff φ_1 and φ_2 hold. The notion of an institution *having implication* is defined similarly.

Theorem 4 (Completeness [9]). 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 [9].

A problem with the above result is that Craig interpolation often fails, e.g. it does not hold for the CASL institution $SubPCFOL^=$ (only for the sublanguage without subsorts and sort-injective signature morphisms, see [8]). 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 [27].

Checking conservativity in the CASL institution The proof rules for refinement are based on an oracle checking 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 [26]. Note that checking conservativeness is at least as complicated as checking non-provability: for a Σ-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.[4]

[4] 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 [20]. 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 undecidable on the basis of our background metatheory $ZFCU$.

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 done in [26]. The main idea is just to mimick 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 [26].

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*. 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.

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

Semantically, a view $v\colon SP_1 \to SP_2$ from a Σ_1-specification SP_1 to a Σ_2-specifiation SP_2 is basically is a *specification morphism* $\sigma\colon SP_1 \to SP_2$, leading to a proof obligation $SP_1 \leadsto SP_2$ **hide** σ. 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* [27, 26], 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.)

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 5. A (fictitious) specification structure for a compiler might look roughly as follows:

$$
\begin{tikzcd}
 & \textsc{Identifier} \arrow[dr] & & \textsc{List} \arrow[dl] & \\
 & & \textsc{ProgramText} \arrow[dl] \arrow[dr] & & \\
 & \textsc{AbstractSyntax} \arrow[dl] \arrow[dr] & & \textsc{SymbolTable} \arrow[dl] \arrow[dr] & \\
\textsc{Parser} \arrow[drr] & & \textsc{StaticAnalyser} \arrow[d] & & \textsc{CodeGenerator} \arrow[dll] \\
 & & \textsc{Compiler} & &
\end{tikzcd}
$$

(The arrows indicate the extension relation between specifications.) An architectural specification of the compiler in CASL might have the following form:

arch spec BUILDCOMPILER =
units I : IDENTIFIER **with sorts** *Identifier*, *Keyword*;
L : ELEM $\rightarrow$ LIST[ELEM];
$IL = L[I$ **fit sort** $Elem \mapsto Identifier]$
$KL = L[I$ **fit sort** $Elem \mapsto Keyword]$
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 to just 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 Σ_1 and Σ_2 and their union $\Sigma_1 \xrightarrow{\iota_1} (\Sigma_1 \cup \Sigma_2) \xleftarrow{\iota_2} \Sigma_2$, for any models $M_1 \in Mod[\Sigma_1]$ and $M_2 \in Mod[\Sigma_2]$, there is at most one model $M \in Mod[\Sigma_1 \cup \Sigma_2]$ such that $M\big|_{\iota_1} = M_1$ and $M\big|_{\iota_2} = M_2$. 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. 7.

Architectural specifications: $ASP ::= \textbf{arch spec } Dcl^* \textbf{ result } T$

Unit declarations: $Dcl ::= U : SP \mid U : SP_1 \xrightarrow{\tau} SP_2$

Unit terms: $T ::= U \mid U[T \textbf{ fit } \sigma] \mid T_1 \textbf{ and } T_2$

Fig. 7. A fragment of the architectural specification formalism

Example 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 $U_1 : \mathtt{SPEC}_1$, then

$U_2 : \mathtt{SPEC}_2$ **given** U_1

abbreviates

$U_2' : \mathtt{SPEC}_1 \to \mathtt{SPEC}_2;$
$U_2 = U_2'[U_1].$

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 $C^{\emptyset}_{st}$. 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.

$$\frac{\vdash \mathit{UDD}^* \rhd C_{st} \qquad C_{st} \vdash T \rhd \Sigma}{\vdash \textbf{arch spec } \mathit{UDD}^* \textbf{ result } T \rhd (C_{st}, \Sigma)}$$

$$\frac{C^{\emptyset}_{st} \vdash \mathit{UDD}_1 \rhd (C_{st})_1 \qquad \cdots \qquad (C_{st})_{n-1} \vdash \mathit{UDD}_n \rhd (C_{st})_n}{\vdash \mathit{UDD}_1 \ldots \mathit{UDD}_n \rhd (C_{st})_n}$$

$$\frac{\vdash \mathit{SP} \rhd \Sigma \qquad U \notin (\mathit{Dom}(P_{st}) \cup \mathit{Dom}(B_{st}))}{(P_{st}, B_{st}) \vdash U : \mathit{SP} \rhd (P_{st}, B_{st} + \{U \mapsto \Sigma\})}$$

$$\frac{\begin{array}{c}\vdash \mathit{SP}_1 \rhd \Sigma_1 \qquad \vdash \mathit{SP}_2 \rhd \Sigma_2 \qquad \tau : \Sigma_1 \to \Sigma_2 \\ U \notin (\mathit{Dom}(P_{st}) \cup \mathit{Dom}(B_{st}))\end{array}}{(P_{st}, B_{st}) \vdash U : \mathit{SP}_1 \xrightarrow{\tau} \mathit{SP}_2 \rhd (P_{st} + \{U \mapsto \tau\}, B_{st})}$$

$$\frac{U \in \mathit{Dom}(B_{st})}{(P_{st}, B_{st}) \vdash U \rhd B_{st}(U)}$$

$$\frac{\begin{array}{c}P_{st}(U) = \tau : \Sigma \to \Sigma' \qquad C_{st} \vdash T \rhd \Sigma_T \qquad \sigma : \Sigma \to \Sigma_T \\ (\tau' : \Sigma_T \to \Sigma'_T, \sigma' : \Sigma' \to \Sigma'_T) \text{ is the pushout of } (\sigma, \tau)\end{array}}{(P_{st}, B_{st}) \vdash U[T \textbf{ fit } \sigma] \rhd \Sigma'_T}$$

$$\frac{\begin{array}{c}C_{st} \vdash T_1 \rhd \Sigma_1 \qquad C_{st} \vdash T_2 \rhd \Sigma_2 \\ \Sigma = \Sigma_1 \cup \Sigma_2 \text{ with inclusions } \iota_1 : \Sigma_1 \to \Sigma, \iota_2 : \Sigma_2 \to \Sigma\end{array}}{(P_{st}, B_{st}) \vdash T_1 \textbf{ and } T_2 \rhd \Sigma}$$

Fig. 8. Static semantics of architectural specifications

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

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.

$$\frac{\vdash \mathit{UDD}^* \Rightarrow \mathcal{C} \qquad \mathcal{C} \vdash T \Rightarrow \mathit{UEv}}{\vdash \textbf{arch spec } \mathit{UDD}^* \textbf{ result } T \Rightarrow (\mathcal{C}, \mathit{UEv})}$$

$$\frac{\mathcal{C}^{\emptyset} \vdash \mathit{UDD}_1 \Rightarrow \mathcal{C}_1 \qquad \cdots \qquad \mathcal{C}_{n-1} \vdash \mathit{UDD}_n \Rightarrow \mathcal{C}_n}{\vdash \mathit{UDD}_1 \ldots \mathit{UDD}_n \Rightarrow \mathcal{C}_n}$$

$$\frac{\vdash \mathit{SP} \Rightarrow \mathcal{M}}{\mathcal{C} \vdash U \colon \mathit{SP} \Rightarrow \mathcal{C} \times \{U \mapsto \mathcal{M}\}}$$

$$\frac{\begin{array}{c}\vdash \mathit{SP}_1 \Rightarrow \mathcal{M}_1 \qquad \vdash \mathit{SP}_2 \Rightarrow \mathcal{M}_2 \\ \mathcal{F} = \{F \colon \mathcal{M}_1 \to \mathcal{M}_2 \mid \text{for } M \in \mathcal{M}_1, F(M)|_{\tau} = M\}\end{array}}{\mathcal{C} \vdash U \colon \mathit{SP}_1 \xrightarrow{\tau} \mathit{SP}_2 \Rightarrow \mathcal{C} \times \{U \mapsto \mathcal{F}\}}$$

$$\frac{}{\mathcal{C} \vdash U \Rightarrow \{E \mapsto E(U) \mid E \in \mathcal{C}\}}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash T \Rightarrow \mathit{UEv} \\ \left.\text{for each } E \in \mathcal{C},\ \mathit{UEv}(E)|_{\sigma} \in \mathit{Dom}(E(U))\right\}(*) \\ \left.\begin{array}{r}\text{for each } E \in \mathcal{C}, \text{ there is a unique } M \in \mathit{Mod}[\Sigma'_T] \text{ such that} \\ M|_{\tau'} = \mathit{UEv}(E) \text{ and } M|_{\sigma'} = E(U)(\mathit{UEv}(E)|_{\sigma})\end{array}\right\}(**) \\ \mathit{UEv}' = \{E \mapsto M \mid E \in \mathcal{C}, M|_{\tau'} = \mathit{UEv}(E), M|_{\sigma'} = E(U)(\mathit{UEv}(E)|_{\sigma})\}\end{array}}{\mathcal{C} \vdash U[T \textbf{ fit } \sigma] \Rightarrow \mathit{UEv}'}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash T_1 \Rightarrow \mathit{UEv}_1 \qquad \mathcal{C} \vdash T_2 \Rightarrow \mathit{UEv}_2 \\ \left.\begin{array}{r}\text{for each } E \in \mathcal{C}, \text{ there is a unique } M \in \mathit{Mod}[\Sigma] \text{ such that} \\ M|_{\iota_1} = \mathit{UEv}_1(E) \text{ and } M|_{\iota_2} = \mathit{UEv}_2(E)\end{array}\right\}(***) \\ \mathit{UEv} = \{E \mapsto M \mid E \in \mathcal{C} \text{ and } M|_{\iota_1} = \mathit{UEv}_1(E), M|_{\iota_2} = \mathit{UEv}_2(E)\}\end{array}}{\mathcal{C} \vdash T_1 \textbf{ and } T_2 \Rightarrow \mathit{UEv}}$$

Fig. 9. Model semantics of architectural specifications

The complete semantics is given in Figures 8 (static semantics) and 9 (model semantics) where we use some auxiliary notation: given a unit context $\mathcal{C}$, 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 Σ-model, and a corresponding condition holds for P_{st}. Obviously, $\mathcal{C}^{\emptyset}$ fits $C^{\emptyset}_{st}$. Now if $\mathcal{C}$ fits C_{st}, then $C_{st} \vdash T \rhd \Sigma$ and $\mathcal{C} \vdash T \Rightarrow \mathit{UEv}$ imply that $\mathit{UEv}(E)$ is a Σ-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. 9. 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 [32]). Under our assumptions, the premise $(**)$ may simply be skipped, as it always holds (since all parameterized units are persistent functions, $E(U)(\mathit{UEv}(E)|_{\sigma})|_{\tau} = \mathit{UEv}(E)|_{\sigma}$, and so the required unique model $M \in \mathit{Mod}[\Sigma'_T]$ 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;
- $SP_{T_1 \textbf{ and } T_2}$ is $(SP_{T_1}$ **and** $SP_{T_2})$;
- $SP_{U[T \textbf{ fit } \sigma]}$ is $((SP_T \textbf{ with } \tau') \textbf{ and } (SP' \textbf{ with } \sigma'))$, where $U : SP \xrightarrow{\tau} SP'$ is the declaration of U and (τ', σ') is the pushout of (σ, τ), 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 $C_{st} \vdash T \rhd \Sigma$ and $\mathcal{C} \vdash T \Rightarrow UEv$ then $\vdash SP_T \rhd \Sigma$ 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 $SP \approx\!\!> SP_T$ **hide** σ.

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

units U : {**sort** s; **op** $a : s$}
ID : {**sort** s; **op** $b : s$} $\rightarrow$ {**sort** s; **op** $b : s$}
F : {**sort** s; **op** $a, b : s$; **axiom** $a = b$} $\rightarrow \ldots$
result F[U **and** ID[U **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.

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* Σ_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 $\vdash SP_G \rhd \Sigma_G$). 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 $U \in Dom(B_v)$, $B_{st}(U) = \Sigma$, where $B_v(U) = i : \Sigma \to \Sigma_G$, and for each $U \in Dom(P_v)$, $P_{st}(U) = \tau$, where $P_v(U) = SP_1 \xrightarrow{\tau} SP_2$.

$$\frac{\vdash Dcl^* :: C_v \qquad C_v \vdash T :: \Sigma \xrightarrow{i} SP'_G \xleftarrow{\theta} SP_G}{\vdash \textbf{arch spec } Dcl^* \textbf{ result } T :: SP'_G \textbf{ hide } i}$$

$$\frac{C_v^{\emptyset} \vdash Dcl_1 :: (C_v)_1 \qquad \cdots \qquad (C_v)_{n-1} \vdash Dcl_n :: (C_v)_n}{\vdash Dcl_1 \ldots Dcl_n :: (C_v)_n}$$

$$\frac{\begin{array}{c} U \notin (Dom(P_v) \cup Dom(B_v)) \qquad \vdash SP \rhd \Sigma \\ (\Sigma_G \xrightarrow{\theta} \Sigma'_G \xleftarrow{i} \Sigma) \text{ is the coproduct of } \Sigma_G \text{ and } \Sigma \end{array}}{\begin{array}{l} (P_v, B_v, SP_G) \vdash U : SP :: \\ \qquad (P_v, (B_v;\theta) + \{U \mapsto i\}, (SP_G \textbf{ with } \theta) \textbf{ and } (SP \textbf{ with } i)) \end{array}}$$

$$\frac{U \notin (Dom(P_v) \cup Dom(B_v))}{(P_v, B_v, SP_G) \vdash U : SP_1 \xrightarrow{\tau} SP_2 :: (P_v + \{U \mapsto SP_1 \xrightarrow{\tau} SP_2\}, B_v, SP_G)}$$

$$\frac{B_v(U) = \Sigma \xrightarrow{i} SP_G}{(P_v, B_v, SP_G) \vdash U :: \Sigma \xrightarrow{i} SP_G \xleftarrow{id} SP_G}$$

$$\frac{\begin{array}{c} (P_v, B_v, SP_G) \vdash T :: \Sigma_T \xrightarrow{i} SP'_G \xleftarrow{\theta} SP_G \\ P_v(U) = SP \xrightarrow{\tau} SP' \quad \vdash SP \rhd \Sigma \quad \vdash SP' \rhd \Sigma' \quad \sigma : \Sigma \to \Sigma_T \\ (\tau' : \Sigma_T \to \Sigma'_T, \sigma' : \Sigma' \to \Sigma'_T) \text{ is the pushout of } (\sigma, \tau) \\ (\tau'' : \Sigma'_G \to \Sigma''_G, i' : \Sigma'_T \to \Sigma''_G) \text{ is the pushout of } (i, \tau') \\ SP \textbf{ with } \sigma;i \rightsquigarrow SP'_G \end{array}}{\begin{array}{l} (P_v, B_v, SP_G) \vdash U[T \textbf{ fit } \sigma] :: \\ \qquad \Sigma'_T \xrightarrow{i'} (SP'_G \textbf{ with } \tau'') \textbf{ and } (SP' \textbf{ with } \sigma';i') \xleftarrow{\theta;\tau''} SP_G \end{array}}$$

$$\frac{\begin{array}{c} (P_v, B_v, SP_G) \vdash T_1 :: \Sigma_1 \xrightarrow{i_1} SP^1_G \xleftarrow{\theta_1} SP_G \\ (P_v, B_v, SP_G) \vdash T_2 :: \Sigma_2 \xrightarrow{i_2} SP^2_G \xleftarrow{\theta_2} SP_G \\ \Sigma = \Sigma_1 \cup \Sigma_2 \text{ with inclusions } \iota_1 : \Sigma_1 \to \Sigma, \iota_2 : \Sigma_2 \to \Sigma \\ (\theta'_2 : \Sigma^1_G \to \Sigma'_G, \theta'_1 : \Sigma^2_G \to \Sigma'_G) \text{ is the pushout of } (\theta_1, \theta_2) \\ j : \Sigma \to \Sigma'_G \text{ satisfies } \iota_1;j = i_1;\theta'_2 \text{ and } \iota_2;j = i_2;\theta'_1 \end{array}}{\begin{array}{l} (P_v, B_v, SP_G) \vdash T_1 \textbf{ and } T_2 :: \\ \qquad \Sigma_1 \cup \Sigma_2 \xrightarrow{j} (SP^1_G \textbf{ with } \theta'_2) \textbf{ and } (SP^2_G \textbf{ with } \theta'_1) \xleftarrow{\theta_1;\theta'_2} SP_G \end{array}}$$

Fig. 10. Verification rules

Given a morphism $\theta : \Sigma_G \to \Sigma'_G$ that extends the global signature (or a global specification morphism $\theta : SP_G \to SP'_G$) we write $B_v;\theta$ for the corresponding 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[5]).

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

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

Architectural specifications yield a specification of the result.

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

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. 10, with diagrams helping to read the more complicated rules for unit application and amalgamation in Fig. 11.

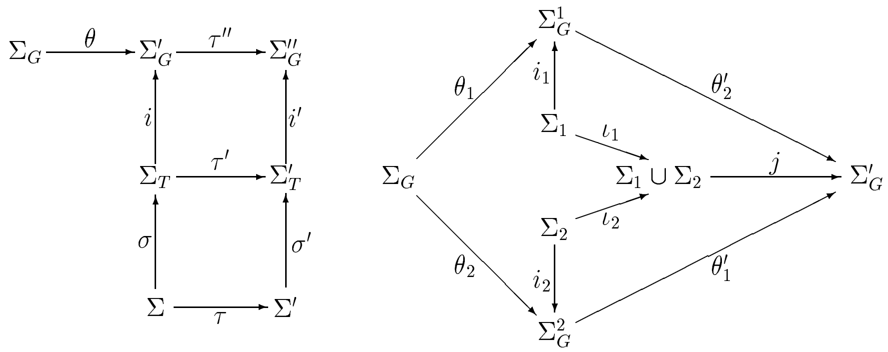

Fig. 11. Diagrams for unit application and amalgamation

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.

[5] More precisely, this is the basic specification consisting of the initial signature with no axioms.

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 $U \in Dom(P_v)$ 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 $U \in Dom(B_v)$, $E(U) = M_G\big|_{B_v(U)}$; 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. 11 (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 [41] for details.

7.3 Enriched CASL 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 6. The simplest case where amalgamation fails is the following: let Σ be the signature with sorts s and t and no operations, and let Σ_1 be the extension of Σ by the subsort relation $s \leq t$. Then the pushout

$$\begin{tikzcd} \Sigma \arrow[r] \arrow[d] & \Sigma_1 \arrow[d] \\ \Sigma_1 \arrow[r] & \Sigma_1 \end{tikzcd}$$

in **SubSig** fails to be amalgamable (since two models of Σ_1, compatible w.r.t. the inclusion of Σ, may interpret the subsort injection differently).

There are two solutions to this problem.

The first 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 [40]. 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.

A second possibility is to replace the global signatures that are used in the static semantics by diagrams of signatures and signature morphisms, see [13]. 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 CASL, these factorization conditions can be dealt with entirely within CASL, using a calculus (the so-called *cell calculus*) for proving equality of morphisms and symbols in the colimit; see [19]. When coming to architectural verification, however, one will still need the embedding into enriched CASL. Development of 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 [4] 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.

CASL's *views* express some aspect of refinement, namely that as a specification is successively refined during the development process, the model class gets smaller and smaller as more and more design decision are made, until a monomorphic design specification or program is reached. However, CASL's 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* [38]. 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 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 [36].

9 CONCLUSION

CASL is a complex specification language providing both a complete formal semantics and a proof calculus for all 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 [28, 39] and of reactive [7, 33, 34, 35] and object-oriented [2, 17] behaviour. *Restrictions* of CASL [25] to sublanguages make it possible to use specialized tool support.

A library of basic datatypes and several case studies have been developed in CASL [1]; they show how CASL works in practice. Of course, tool support is an important issue as well. There is a CASL tool set [23] providing a parser and static analysis for the various levels of CASL. Concerning proof support for basic specifications, the CASL tool set provides several logical codings that interface CASL to existing theorem provers such as Isabelle/HOL. The tool MAYA [5] provides management of proof obligations arising from CASL structured specifications. These proof obligations can be discharged by calling external theorem provers. Currently, this tool support is also being made independent of the underlying logical system, so that the orthogonality of the different levels of CASL is also reflected at the level of tools. Then programming languages (formalized as particular institutions) can also be integrated, leading to a framework and environment for formal software development.

REFERENCES

[1] CASL case studies. Available at `http://www.pst.informatik.uni-muenchen.de/~baumeist/CoFI/case.html`.

[2] 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.

[3] 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.

[4] E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner. *Algebraic Foundations of Systems Specification*. Springer, 1999.

[5] S. Autexier and T. Mossakowski. Integrating HOLCASL into the development graph manager MAYA. In A. Armando, editor, *Frontiers of Combining Systems, 4th International Workshop*, volume 2309 of *Lecture Notes in Computer Science*, pages 2–17. Springer-Verlag, 2002.

[6] J. Barwise and J. Etchemendy. *Language, proof and logic*. CSLI publications, 2002.

[7] 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.

[8] T. Borzyszkowski. Generalized interpolation in CASL. *Information Processing Letters*, 76/1-2:19–24, 2000.

[9] T. Borzyszkowski. Logical systems for structured specifications. *Theoretical Computer Science*, 286:197–245, 2002.

[10] 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.

[11] CoFI. The Common Framework Initiative for algebraic specification and development, electronic archives. Notes and Documents accessible from `http://www.brics.dk/Projects/CoFI/`.

[12] CoFI Language Design Task Group. Casl – The CoFI Algebraic Specification Language – Summary. Documents/CASL/Summary, in [11], Mar. 2001.

[13] CoFI Semantics Task Group. Casl – The CoFI Algebraic Specification Language – Semantics. Note S-9 (Documents/CASL/Semantics, version 0.96), in [11], July 1999.

[14] 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.

[15] 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.

[16] H. Herrlich and G. Strecker. *Category Theory*. Allyn and Bacon, Boston, 1973.

[17] H. Hussmann, M. Cerioli, and H. Baumeister. From UML to Casl (static part). Technical report, 2000. Technical Report of DISI - Universit di Genova, DISI-TR-00-06, Italy.

[18] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1990.

[19] 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.

[20] K. Meinke and J. V. Tucker, editors. *Many-sorted Logic and its Applications*. Wiley, 1993.

[21] J. Meseguer. General logics. In *Logic Colloquium 87*, pages 275–329. North Holland, 1989.

[22] 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.

[23] T. Mossakowski. Casl: From semantics to tools. In S. Graf and M. Schwartzbach, editors, *TACAS 2000*, volume 1785 of *Lecture Notes in Computer Science*, pages 93–108. Springer-Verlag, 2000.

[24] 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.

[25] T. MOSSAKOWSKI. RELATING CASL WITH OTHER SPECIFICATION LANGUAGES: the institution level. *Theoretical Computer Science*, 286:367–475, 2002.

[26] T. Mossakowski, S. Autexier, and D. Hutter. CASL proof calculus. To appear in [30].

[27] 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.

[28] 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.

[29] 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.

[30] P. D. Mosses (ed.). CASL— *the common algebraic specification language*. Springer. To appear.

[31] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*. Oxford Univ. Press, 1990.

[32] L. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996. 2nd edition.

[33] 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.

[34] 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.

[35] M. Roggenbach. CSP-CASL – a new integration of process algebra and algebraic specification. Manuscript, Bremen.

[36] D. SANNELLA AND A. TARLECKI. ON OBSERVATIONAL EQUIVALENCE AND ALGEBRAIC SPECIFICATION. *Journal of Computer and System Sciences*, 34:150–178, 1987.

[37] D. SANNELLA AND A. TARLECKI. SPECIFICATIONS IN AN ARBITRARY INSTITUTION. *Information and Computation*, 76:165–210, 1988.

[38] D. SANNELLA AND A. TARLECKI. TOWARD FORMAL DEVELOPMENT OF PROGRAMS FROM ALGEBRAIC SPECIFICATIONS: Implementations revisited. *Acta Informatica*, 25:233–281, 1988.

[39] 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.

[40] L. Schröder, T. Mossakowski, P. Hoffman, B. Klin, and A. Tarlecki. Amalgamation in the semantics of CASL. Theoretical Computer Science, to appear.

[41] A. TARLECKI. ABSTRACT SPECIFICATION THEORY: an overview. In M. Broy and M. Pizka, editors, *Models, Algebras, and Logics of Engineering Software*, NATO Sci-

ence Series — Computer and System Sciences. IOS Press, 2003. to appear.