Under consideration for publication in Math. Struct. in Comp. Science

Observational Interpretation of Casl Specifications

MICHEL BIDOIT[†], DONALD SANNELLA[‡] and ANDRZEJ TARLECKI[§]

[†] *Laboratoire Spécification et Vérification, CNRS & ENS de Cachan, France*
Website: `www.lsv.ens-cachan.fr/~bidoit/`
[‡] *Laboratory for Foundations of Computer Science, University of Edinburgh, UK*
Website: `homepages.inf.ed.ac.uk/dts/`
[§] *Institute of Informatics, Warsaw University, and*
Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland
Website: `www.mimuw.edu.pl/~tarlecki/`

Received 12 September 2006; Revised 15 June 2007

The way that refinement of individual "local" components of a specification relates to development of a "global" system from a specification of requirements is explored. Observational interpretation of specifications and refinements add expressive power and flexibility while bringing in some subtle problems. Our study of these issues is carried out in the context of Casl architectural specifications. We introduce a definition of observational equivalence for Casl models, leading to an observational semantics for architectural specifications for which we prove important properties. Overall, this fulfills the long-standing goal of complementing the standard semantics of Casl specifications with an observational view that supports observational refinement of specifications in combination with Casl-style architectural design.

1. Introduction

There has been a great deal of work in the algebraic specification tradition on formalizing the rather intuitive and appealing idea of program development by stepwise refinement, including (Ehrig *et al.* 1982; Ganzinger 1983; Schoett 1987; Sannella and Tarlecki 1988b); for a survey, see (Ehrig and Kreowski 1999). There are many issues that make this a difficult problem, and some of them are rather subtle, one example being the relationship between specification structure and program structure, and another being the tradeoff between the expressive power of a specification formalism and the ease of reasoning about specifications. Significant complications result when "observational" or "behavioural" aspects of specifications are considered, whereby the definition of correctness takes into account only the results of those computations that can be directly observed. An overview that covers most of our own contributions is (Sannella and Tarlecki 1997), with some more recent work addressing the problem of how to prove correctness of refinement steps (Bidoit and Hennicker 1998; Bidoit and Hennicker 2006), the design of a convenient formalism for writing specifications (Bidoit, Sannella and Tarlecki 2002a; Astesiano *et* *al.* 2002; CoFI 2004), and applications to data refinement in typed λ-calculus (Honsell *et al.* 2000).

A new angle that we explore here is the "global" effect of refining individual "local" components of a specification. This involves a well-known technique from algebraic specification, namely the use of pushouts of signatures and amalgamation of models to build large systems by composition of separate interrelated components. The situation becomes considerably more subtle when observational interpretation of specifications and refinements is brought into the picture.

Part of the answer has already been provided, the main references being Schoett's thesis (Schoett 1987; Schoett 1990) and our work on formal development in the EXTENDED ML framework (Sannella and Tarlecki 1989); the general ideas go back at least to (Hoare 1972). We have another look at these issues here, in the context of the CASL specification formalism (Astesiano *et al.* 2002; CoFI 2004) and in particular, its *architectural specifications* (Bidoit, Sannella and Tarlecki 2002a). Architectural specifications, for describing the modular structure of software systems, are probably the most novel feature of CASL. We view them here as a means of making complex refinement steps, by defining a construction to be used to build the overall system from implementations of individually-specified units; these may include parametrized units that contribute to this construction.

This paper combines and expands on previous work reported in (Bidoit, Sannella and Tarlecki 2002a; Bidoit, Sannella and Tarlecki 2002b; Bidoit, Sannella and Tarlecki 2004; Baumeister *et al.* 2004; Schröder *et al.* 2005). It interweaves three strands. The first strand (Sects. 2 and 5) recalls the basic semantic concepts of CASL and introduces observational equivalence for CASL models and the induced observational interpretation of CASL basic and structured specifications. In contrast to (Bidoit, Sannella and Tarlecki 2002b), true CASL models are considered rather than standard many-sorted total algebras.

A second strand (Sects. 3 and 6) explores the use of local constructions in an arbitrary global context, and its interaction with an observational view of requirements specifications. In particular, stability and observational correctness of constructions on CASL models are treated, and practical local criteria to establish both properties are formulated.

The final strand (Sects. 4 and 7) provides a careful analysis of the semantics of CASL architectural specifications, taking account of the fact that amalgamability is not ensured for CASL models and linking with the other strands to provide such specifications with an observational semantics. Key invariant properties of the semantics are precisely formulated and proved.

Due to space considerations we do not deal with full-blown CASL as defined in (Mosses 2004), but the addition of unit definitions to the treatment in (Bidoit, Sannella and Tarlecki 2002b) together with a proper account of dependencies between units means that the extension to full CASL would be routine. The analysis of invariants linking the static semantics and model semantics of architectural specifications in Sect. 4 provides essential insight into the semantics of full CASL that was implicit in (Baumeister *et al.* 2004); this reiterates Thm. 2 in (Schröder *et al.* 2005) and provides a basis for an analogous treatment of the observational case in Sect. 7.

An orthogonal view of the structure of this paper is that Sects. 2–4 present a standard treatment of CASL basic and structured specifications, local constructions and their use in a global context, and CASL architectural specifications; a comprehensive observational treatment is then given in Sects. 5–7. An example in Sect. 8, based on one in (Bidoit, Sannella and Tarlecki 2004), provides a concrete illustration of some of the points that arise.

Overall, this fulfills the long-standing goal of complementing the standard semantics of CASL specifications (Baumeister *et al.* 2004) with an observational view that supports observational refinement of specifications in combination with CASL-style architectural design.

2. CASL **Institution and Specifications**

A basic assumption underpinning algebraic specification and derived approaches to software specification and development is that programs are modelled as algebras (of some kind) with their "types" captured by algebraic signatures (again, adapted as appropriate). Then specifications include axioms describing the required properties. This leads to quite a flexible framework, which can be tuned as desired to cope with various programming features of interest by selecting the appropriate variation of algebra, signature and axiom. This flexibility has been formalized via the notion of *institution* (Goguen and Burstall 1992) and related work on the theory of specifications and formal program development (Sannella and Tarlecki 1988a; Sannella and Tarlecki 1997; Bidoit and Hennicker 1993).

Let us recall that an institution defines a notion of signature together with for any signature Σ, a set of Σ-sentences, a class of Σ-models equipped with homomorphisms, and a satisfaction relation between Σ-models and Σ-sentences. Moreover, signatures come equipped with signature morphisms, forming a category. Any signature morphism induces a translation of sentences and a translation of models (the latter going in the opposite direction to the morphism). All this can be expressed very concisely using the language of category theory: we require a category $\mathbf{Sig}$, a functor $\mathbf{Sen} : \mathbf{Sig} \to \mathbf{Set}$, a (contravariant) functor $\mathbf{Mod} : \mathbf{Sig}^{op} \to \mathbf{Cat}$, and a family of binary relations $\langle \models_\Sigma \subseteq |\mathbf{Mod}(\Sigma)| \times \mathbf{Sen}(\Sigma)\rangle_{\Sigma\in|\mathbf{Sign}|}$. The only semantic requirement is that when we change signatures using a signature morphism, the induced translations of sentences and of models preserve the satisfaction relation.

By now it is standard to base work on specification languages and formal program development on the notion of an institution, so that a clear separation between logic-dependent details and general logic-independent aspects of the work can be achieved. We follow this below, recalling the logical system of CASL (Bidoit and Mosses 2004).

CASL is an algebraic specification language for describing CASL *models*: many-sorted algebras with subsorts, partial and total operations, and predicates. CASL models are classified by CASL *signatures*, which give *sort* names (with their subsorting relation), partial and total *operation* names, and *predicate* names, together with *profiles* of operations and predicates. In CASL models, subsorts and supersorts are linked by implicit *subsort embeddings* that are required to compose with each other and to be compatible with operations and predicates with the same names.

Recalling (and slightly simplifying) some technical detail from (Baumeister *et al.* 2004): a CASL signature is a tuple $\Sigma = (S, TF, PF, P, \leq)$, where S is a set of sort names, $TF = \langle TF_{ws}\rangle_{ws\in S^+}$ and $PF = \langle PF_{ws}\rangle_{ws\in S^+}$ are families of total and partial, respectively, operation names, indexed by their profiles (which consist of their arity $w \in S^*$ and result sort $s \in S$), $P = \langle P_w\rangle_{w\in S^*}$ is a family of predicate names, indexed by their arities, and $\leq$ is a subsorting preorder on S (a relation that is reflexive and transitive). For simplicity, we bluntly assume that no overloading is allowed, that is, that all the sets in TF, PF, and P are mutually disjoint[1]. We write $f\colon s_1 \times \cdots \times s_n \to s$ when $s_1, \ldots, s_n, s \in S$ and $f \in TF_{s_1\ldots s_n s}$; similar notation is used for partial operation names and for predicate symbols. If $n = 0$ then f is a constant and we write $f\colon s$. For CASL signatures $\Sigma = (S, TF, PF, P, \leq)$ and $\Sigma' = (S', TF', PF', P', \leq')$, a morphism between them, written $\sigma : \Sigma \to \Sigma'$, maps sort names in S to sort names in S' so that the subsorting preorder is preserved, operation names in $TF \cup PF$ to operation names in $TF' \cup PF'$ so that their totality and profiles are preserved, and predicate names in P to predicate names in P' so that their arities are preserved. This yields a category **Sig** of CASL signatures and their morphisms with the obvious identities and component-wise composition.

Given a CASL signature $\Sigma = (S, TF, PF, P, \leq)$, we define its expansion to a many-sorted signature $\Sigma^{\#}$ that retains the set of sorts S and includes the operation and predicate names from TF, PF and P, adding for all $s \leq s'$ in Σ, a new total operation name $em^{s\leq s'}\colon s \to s'$ for subsort embedding, a new partial operation name $pr^{s\leq s'}\colon s' \to s$ for subsort projection, and a new predicate name $in^{s\leq s'}\colon s'$ for subsort membership. Note that $(\_\_)^{\#}$ extends to signature morphisms in an obvious way.

Now, a CASL model over the CASL signature $\Sigma = (S, TF, PF, P, \leq)$ is a structure M over the signature $\Sigma^{\#}$, which consists of a carrier set $|M|_s$ for each sort $s \in S$, a (partial) function $f_M\colon |M|_{s_1} \times \cdots \times |M|_{s_n} \to |M|_s$ for each operation name $f\colon s_1 \times \cdots \times s_n \to s$ in $\Sigma^{\#}$ (with f_M being total for total operation names f) and a relation $p_M \subseteq |M|_{s_1} \times \cdots \times |M|_{s_n}$ for each predicate name $p\colon s_1 \times \cdots \times s_n$, such that for all $s \leq s'$ in Σ, the subsort embedding $em_M^{s\leq s'}\colon |M|_s \to |M|_{s'}$ is injective, the subsort projection $pr_M^{s\leq s'}\colon |M|_{s'} \to |M|_s$ is defined exactly on the image of $em_M^{s\leq s'}$ as its inverse, and the subsort membership predicate $in_M^{s\leq s'} \subseteq |M|_{s'}$ holds exactly on the image of $em_M^{s\leq s'}$. Moreover, we require that $em_M^{s\leq s}$ is the identity for $s \in S$, and that the embeddings compose, that is, if $s \leq s' \leq s''$ then $em_M^{s\leq s''}$ is the composition of $em_M^{s\leq s'}$ and $em_M^{s'\leq s''}$.

This yields the class of CASL Σ-models, which form a category $\mathbf{Mod}(\Sigma)$ with homomorphisms between $\Sigma^{\#}$-structures defined as usual, as maps that preserve predicates as well as the definedness and values of operations. A homomorphism is *strong* if it also reflects the predicates and the definedness of operations. Given a CASL Σ-model M, a

[1] This assumption is unrealistic in practical examples, especially when subsorting is involved; CASL deals with this properly, imposing only a considerably weaker version of this restriction. The issues that arise are irrelevant for the topic of this paper.

submodel is any CASL Σ-model N with carriers of N included in those of M such that the inclusion function $|N| \hookrightarrow |M|$ is a strong homomorphism, cf. closed subalgebras in (Burmeister 1986).

As expected, kernels of homomorphisms between CASL models are *congruences*: equivalence relations on model carriers closed under operations when defined in the model (this also applies to the subsort embeddings and projections). Kernels of strong homomorphisms are *strong congruences*: these are congruences that in addition preserve predicates and definedness of operations. Given any CASL Σ-model M and congruence $\simeq$ on it, the quotient of M by $\simeq$ is defined as the quotient of M as a $\Sigma^{\#}$-structure by $\simeq$; it is easy to check that the usual definition yields a $\Sigma^{\#}$-structure which is a CASL Σ-model, and that the natural quotient homomorphism is strong whenever the congruence $\simeq$ is strong.

Any CASL signature morphism $\sigma\colon \Sigma \to \Sigma'$ determines a *reduct* functor from $\mathbf{Mod}(\Sigma')$ to $\mathbf{Mod}(\Sigma)$, where for any Σ'-model $M' \in |\mathbf{Mod}(\Sigma')|$, its reduct $M'|_{\sigma} \in |\mathbf{Mod}(\Sigma)|$ is defined as the $\sigma^{\#}$-reduct of the $(\Sigma')^{\#}$-structure M': any sort, operation or predicate name ν in $\Sigma^{\#}$ gets the same interpretation in $M'|_{\sigma}$ as $\sigma^{\#}(\nu)$ has in M'. Similarly for homomorphisms, and for arbitrary relations between carriers of CASL models. This completes the definition of a functor $\mathbf{Mod} : \mathbf{Sig}^{op} \to \mathbf{Cat}$.

It is easy to check that the category **Sig** of CASL signatures is (finitely) cocomplete, with colimits of diagrams given in the expected, component-wise way. Note in particular that the subsort preorder in the colimit signature is the transitive closure of the union of the images of the subsort preorders of the signatures in the diagram under the colimit injections. We will assume that some standard construction of pushouts in **Sig** is given.

Colimits in **Sig** offer a rudimentary way of putting together CASL signatures and basic specifications over them (see below), very much as in the standard algebraic framework (Ehrig and Mahr 1985). When it comes to model theory though, things are more difficult, since CASL does not ensure that the *amalgamation property* holds.

Definition 2.1 (Amalgamation). A pushout in the category of CASL signatures:

$$\begin{tikzcd} \Sigma_1 \arrow[r, "\iota'"] & \Sigma_1' \\ \Sigma \arrow[u, "\gamma"] \arrow[r, "\iota"'] & \Sigma' \arrow[u, "\gamma'"'] \end{tikzcd}$$

ensures amalgamability if for all models $M_1 \in |\mathbf{Mod}(\Sigma_1)|$ and $M' \in |\mathbf{Mod}(\Sigma')|$ such that $M_1|_{\gamma} = M'|_{\iota}$ there exists a unique model $M_1' \in |\mathbf{Mod}(\Sigma_1')|$ such that $M_1'|_{\iota'} = M_1$ and $M_1'|_{\gamma'} = M'$. We sometimes write $M_1 \oplus M'$ for such a unique M_1' and call it the *amalgamation of* M_1 *and* M', when the pushout is clear from the context.

When the signature morphism ι is given and the pushout as above ensures amalgamability, we will refer to the morphism γ as *admissible* (cf. Def. 3.3 below).

It is worth stressing that pushouts of CASL signature morphisms between signatures with no proper subsorts (i.e., the subsorting preorders are identities) always ensure amalgamability. The potential problems are caused by the built-in requirements of uniqueness and composability of subsort embeddings in CASL models. The simplest example of a pushout that does not ensure amalgamability is when Σ contains just two sorts, and both Σ_1 and Σ' expand Σ by adding a new subsort relationship between the two sorts. The pushout signature then coincides with $\Sigma_1 = \Sigma'$ (and so allows for one subsort embedding between the two sorts), and two models over Σ_1 and Σ' with common Σ-reduct amalgamate only if they happen to share the same subsort embedding. Perhaps surprisingly, the problem whether a pushout (or more generally, a colimit) ensures amalgamability is in general undecidable, but a number of effective algorithms to determine this in various practically relevant cases can be given. However, we do not know any easy syntactic condition that would ensure amalgamability without excluding some cases that naturally arise in practical specifications. For instance, requiring that ι and γ in the diagram above do not introduce new subsorting relationships between sorts from Σ is not sufficient. To see this, consider Σ with just two independent sorts, Σ_1 which adds a new common subsort for them, and Σ' which add a new common supersort for them. Then the resulting pushout does not ensure amalgamability. We refer to (Schröder *et al.* 2001; Klin *et al.* 2001; Schröder *et al.* 2005) for further examples and a more complete study of amalgamability in CASL. Here, we just guard any use of amalgamation with a requirement that the relevant pushout ensures amalgamability.

In the framework of CASL, if a pushout ensures amalgamability (of CASL models, as above) then it also ensures amalgamability of homomorphisms:

Lemma 2.2. Suppose that the following pushout

$$\begin{tikzcd} \Sigma_1 \arrow[r, "\iota'"] & \Sigma'_1 \\ \Sigma \arrow[u, "\gamma"] \arrow[r, "\iota"'] & \Sigma' \arrow[u, "\gamma'"'] \end{tikzcd}$$

ensures amalgamability. Then for all homomorphisms $h_1 \colon M_1 \to N_1$ in $\mathbf{Mod}(\Sigma_1)$ and $h' \colon M' \to N'$ in $\mathbf{Mod}(\Sigma')$ such that $h_1|_\gamma = h'|_\iota$ there exists a unique homomorphism $h'_1 \colon M'_1 \to N'_1$ in $\mathbf{Mod}(\Sigma'_1)$ such that $h'_1|_{\iota'} = h_1$ and $h'_1|_{\gamma'} = h'$. Moreover, h'_1 is strong if both h_1 and h' are strong.

Proof. Let $M'_1 = M_1 \oplus M'$ and $N'_1 = N_1 \oplus N'$ (they are well-defined, since the pushout ensures amalgamability). For each sort s_1 in Σ_1, put $(h'_1)_{\iota'(s_1)} = (h_1)_{s_1}$; for each sort s' in Σ', put $(h'_1)_{\gamma'(s')} = (h')_{s'}$. By the construction of pushouts in **Sig**, this yields a well-defined family of functions $(h'_1)_s \colon |M'_1|_s \to |N'_1|_s$, for sorts s in Σ'_1. The required compatibility with the predicates and operations of the form $(\iota')^\#(f_1)$, for f_1 in $\Sigma_1^\#$, follows from the compatibility of h_1 with the predicates and operations in $\Sigma_1^\#$; similarly for the predicates and operations of the form $(\gamma')^\#(f')$ for f' in $(\Sigma')^\#$. Consider then a subsort embedding in $(\Sigma'_1)^\#$. Since the subsort relation in Σ'_1 is the transitive closure of the union of the images of the subsort relations in Σ_1 and Σ' under ι' and γ', respectively, the embedding is a composition of embedding operations of the forms considered above — and so compatibility follows by an easy induction. Similarly for subsort projections in

Σ'_1, and then for the subsort membership predicates (which are defined as the domains of the corresponding subsort projections). □

Given a CASL signature Σ, we assume the usual definition of a first-order formula (with quantification and the usual logical connectives) built over *atomic formulae* which include strong and existential equalities, definedness formulae and predicate applications, over the many-sorted signature $\Sigma^{\#}$, and its satisfaction in a $\Sigma^{\#}$-structure. Adding so-called generation constraints as special, non-first-order sentences, yields the set of CASL Σ-sentences, written $\mathbf{Sen}(\Sigma)$. Given a CASL signature morphism $\sigma: \Sigma \to \Sigma'$, the translation of any Σ-sentence $\varphi \in \mathbf{Sen}(\Sigma)$ is defined as usual, and we write it as $\sigma(\varphi)$, see (Baumeister *et al.* 2004). This defines a functor $\mathbf{Sen} : \mathbf{Sig} \to \mathbf{Set}$.

As usual for first-order logic, satisfaction is defined for the more general case of formulae with free variables; we write $M[v] \models_\Sigma \varphi$ to state that the Σ-formula φ with free variables in a set X holds in the Σ-model M under the valuation $v: X \to |M|$. The signature subscript in $\models_\Sigma$ is usually left implicit. The notation $(t)_{M[v]}$ is used to denote the value of a term t with variables in X in the model M under the valuation $v: X \to |M|$; this may be undefined, when the term involves partial operations. Satisfaction of formulae and evaluation of terms only depend on the valuation of their free variables. We drop the valuation v in this notation for closed terms (terms with no variables) and sentences (formulae with no free variables). The satisfaction of sentences is preserved under signature morphisms: for any $\sigma: \Sigma \to \Sigma'$, $M' \in |\mathbf{Mod}(\Sigma')|$ and $\varphi \in \mathbf{Sen}(\Sigma)$, we have

$$M'|_\sigma \models \varphi \iff M' \models \sigma(\varphi)$$

We consider CASL formulae built over the usual algebraic terms only, so in particular CASL conditional terms are excluded (they can be easily eliminated in formulae anyway, see (CoFI 2004)).

We introduce a more general form of conditional terms, as follows, but without allowing them in formulae. Given a CASL signature Σ, a *conditional term* of sort s with variables in X is of the form $c = \langle(\phi_i, t_i)\rangle_{i \geq 0}$, where for $i \geq 0$, ϕ_i are formulae with variables in X, and t_i are terms of sort s with variables in X. Given a Σ-model M and a valuation $v: X \to |M|$, the value $c_{M[v]}$ of such a conditional term c is $(t_k)_{M[v]}$ for the least $k \geq 0$ such that $M[v] \models \phi_k$, or is undefined if no such $k \geq 0$ exists. Note that the infinitary unfolding of any recursive definition can be captured by such a conditional term. Therefore we use these conditional terms to model arbitrary computations, even though they go well beyond what programming languages offer: arbitrary formulae are used as conditions without regard to decidability, the sequence of conditions and terms need not even be recursively enumerable, etc. Some of this generality will be excluded by requirements arising from discussion in Sections 5 and 6.1.

We use these conditional terms to generalize derived signature morphisms (Goguen, Thatcher and Wagner 1978). A *derived signature morphism* $\delta: \Sigma \to \Sigma'$ maps partial operation symbols $f: s_1 \times \ldots \times s_n \to s$ in Σ to conditional Σ'-terms of sort $\delta(s)$ with variables $\{x_1: \delta(s_1), \ldots, x_n: \delta(s_n)\}$. Evidently, such a derived signature morphism $\delta: \Sigma \to \Sigma'$ still determines a reduct function $\_|_\delta$ mapping Σ'-models to Σ-models. In general this does *not* extend to a reduct functor between model categories, since values of conditional terms with arbitrary conditions need not be preserved by homomorphisms (but see the comment following Lemma 5.5).

The basic level of CASL includes *declarations* to introduce components of signatures and *axioms* to give properties that characterize *models* of a specification. Consequently, a basic CASL specification SP amounts to a definition of a signature Σ and a set of axioms $\Phi \subseteq \mathbf{Sen}(\Sigma)$. It denotes the class $[\![SP]\!] \subseteq |\mathbf{Mod}(\Sigma)|$ of SP-models, which are those Σ-models that *satisfy* all the axioms in Φ:

$$[\![SP]\!] = \{M \in |\mathbf{Mod}(\Sigma)| \mid M \models \Phi\}.$$

Apart from basic specifications as above, CASL provides ways of building complex specifications out of simpler ones by means of various *structuring constructs*. These include translation, hiding, union, and both free and loose forms of extension. *Generic specifications* and their *instantiations* with pushout-style semantics (Burstall and Goguen 1980; Ehrig and Mahr 1985) are also provided. Structured specifications built using these constructs are given a compositional semantics where each specification SP determines a signature $Sig[SP]$ and a class $[\![SP]\!] \subseteq |\mathbf{Mod}(Sig[SP])|$ of models. Most of the details, given in (Baumeister *et al.* 2004), are irrelevant for the purposes of this paper. It is enough to know the following: for any specification SP and signature morphism $\sigma\colon Sig(SP) \to \Sigma'$, we write SP **with** σ for the translation of SP along σ, with semantics given by $Sig[SP \textbf{ with } \sigma] = \Sigma'$ and $[\![SP \textbf{ with } \sigma]\!] = \{M' \in |\mathbf{Mod}(\Sigma')| \mid M'|_\sigma \in [\![SP]\!]\}$, and for any two specifications SP_1 and SP_2 with common signature, we write SP_1 **and** SP_2 for their union, with semantics given by $Sig[SP_1 \textbf{ and } SP_2] = Sig[SP_1] = Sig[SP_2]$ and $[\![SP_1 \textbf{ and } SP_2]\!] = [\![SP_1]\!] \cap [\![SP_2]\!]$. Note that union in CASL generalizes this by allowing $Sig[SP_1] \neq Sig[SP_2]$.

3. Software Components and Their Correctness

The intended use of CASL, as of any such specification formalism, is to specify programs. Each CASL specification should determine a class of programs that correctly realize the specified requirements. To fit this into the formal view of CASL specifications, programs must be written in a programming language having a semantics which assigns to each program its *denotation* as a CASL model.[2] Then each program P determines a CASL signature $Sig[P]$ and a model $[\![P]\!] \in |\mathbf{Mod}(Sig[P])|$. Any specification SP is then a description of its admissible realizations: a program P is a *(correct) realization* of SP if $Sig[P] = Sig[SP]$ and $[\![P]\!] \in [\![SP]\!]$.

Let us now consider component-based systems, that is, systems obtained by assembling components, rather than "monolithic" programs. We take a rather restrictive view of components, namely *software components* (understood as pieces of code) in contrast with

[2] This may be rather indirect, and in general involves a non-trivial abstraction step. It has not yet been attempted for any real programming language, but see (Schröder and Mossakowski 2002) for an outline of how this could be done for Haskell. See also the pre-CASL work on Extended ML (Kahrs, Sannella and Tarlecki 1997), and see Larch (Guttag and Horning 1993) for another attempt to link a specification language with various programming languages.

system components (understood as self-contained processors with their own hardware and software interacting with each other and the environment by exchanging messages across linking interfaces). However, our view is consistent with the best accepted definition in the software industry, see (Szyperski 1998): a (software) component is an independently-deployable unit of composition with contractually specified interfaces and fully explicit context dependencies.

To capture this, we will consider that a software component ΔP determines a "parameter" signature, say Σ, corresponding to the symbols required by the component, and a "result" signature, say Σ', corresponding to the symbols provided by the component, together with a signature morphism $\iota\colon \Sigma \to \Sigma'$ relating the "parameter" signature to the "result" signature. Thereby $\iota\colon \Sigma \to \Sigma'$ corresponds to the (syntactic part of the) interface of the software component.

Then the software component ΔP determines a function $F = [\![\Delta P]\!]$ from CASL Σ-models to CASL Σ'-models. This function may be partial, see below. When assembled with (applied to) a sub-system P (determining a CASL signature $\mathit{Sig}[P] = \Sigma$ and a model $[\![P]\!] \in |\mathbf{Mod}(\Sigma)|$), the software component ΔP "extends" P to a larger system, say $\Delta P(P)$, with signature $\mathit{Sig}[\Delta P(P)] = \Sigma'$, determining a CASL model $[\![\Delta P(P)]\!] \in |\mathbf{Mod}(\Sigma')|$. It is intuitively clear that the software component "preserves" the sub-system it is applied to, so $[\![\Delta P(P)]\!]\big|_\iota = [\![P]\!]$.

Thus a software component ΔP determines a semantic object F called a *local construction* according to the definition below. Since software components preserve their arguments, we assume that such constructions are *persistent*: the argument of a construction is always fully included in its result, without modification[3] — note that this assumption holds for all constructions that can be declared and specified in CASL, see Sect. 4. In fact, we generalize CASL somewhat by considering arbitrary signature morphisms rather than just inclusions.

Definition 3.1 (Local construction). Given a signature morphism $\iota\colon \Sigma \to \Sigma'$, a *local construction along* ι is a persistent partial function $F\colon |\mathbf{Mod}(\Sigma)| \rightharpoonup |\mathbf{Mod}(\Sigma')|$ (for each $M \in \mathit{dom}(F)$, $F(M)\big|_\iota = M$). We write $\mathbf{Mod}(\Sigma \xrightarrow{\iota} \Sigma')$ for the class of all local constructions along ι.

We will not dwell here on how particular local constructions are defined. Free functor semantics for parametrized specifications is one way to proceed, with the persistency requirement giving rise to additional proof obligations (Ehrig and Mahr 1985). Perhaps closer to ordinary programming, any "definitional" derived signature morphism $\delta\colon \Sigma' \to \Sigma$ that defines Σ'-components in terms of Σ-components naturally gives rise to a local

[3] Otherwise we would have to explicitly indicate "sharing" between the argument and result of each construction, and explain how such sharing is preserved by the various ways of putting together constructions, as was painfully spelled out in (Sannella and Tarlecki 1989). If necessary, superfluous components of models constructed using persistent constructions can be discarded at the end using the reduct along a signature inclusion.

construction, since the induced reduct function $\_|_\delta\colon |\mathbf{Mod}(\Sigma)| \to |\mathbf{Mod}(\Sigma')|$ is a local construction along a signature morphism $\iota\colon \Sigma \to \Sigma'$ whenever $\iota;\delta = id_\Sigma$.[4]

Of course we are interested in specifications of software components, that is, in "semantic" specifications of the parameter required by the component and of its result (and not only in the "syntactic" specification of its interface given by $\iota\colon \Sigma \to \Sigma'$). Thus, in our algebraic setting, we will specify a software component by a pair of specifications SP and SP', written $SP \xrightarrow{\iota} SP'$, where SP specifies the symbols and their properties required by the component, SP' specifies the symbols and the properties provided by the component, together with a signature morphism $\iota\colon Sig[SP] \to Sig[SP']$ relating the parameter signature to the result signature. Indeed we require ι to be a *specification morphism* $\iota\colon SP \to SP'$, i.e., for all $M' \in [\![SP']\!]$, $M'|_\iota \in [\![SP]\!]$. This amounts to demanding that the result specification SP' includes the properties of the parameter required by the parameter specification SP. The fact that the result actually has those properties is guaranteed by the persistency of the local construction.

The following definition states when a local construction F, determined by a software component ΔP, is a correct realization of a given component specification. (We refer to this as *literal* correctness by contrast with the *observational* correctness of Def. 6.9 below.)

Definition 3.2 (Literal correctness). A local construction F along $\iota\colon Sig[SP] \to Sig[SP']$ is *literally correct w.r.t. SP and SP′* if for all models $M \in [\![SP]\!]$, $M \in dom(F)$ and $F(M) \in [\![SP']\!]$. We write $[\![SP \xrightarrow{\iota} SP']\!]$ for the class of all local constructions along ι that are literally correct w.r.t. SP and SP'.

Hence to realize the component specification $SP \xrightarrow{\iota} SP'$, we should provide a software component ΔP that extends any realization P of SP to a realization $P' = \Delta P(P)$ of SP'. The basic semantic property required is that for all programs P such that $[\![P]\!] \in [\![SP]\!]$, $\Delta P(P)$ is a program that extends P and realizes SP' (semantically: $[\![\Delta P(P)]\!]|_\iota = [\![P]\!]$ and $[\![\Delta P(P)]\!] \in [\![SP']\!]$). This amounts to requiring that the partial function $F \in \mathbf{Mod}(\Sigma \xrightarrow{\iota} \Sigma')$ determined by ΔP preserves its argument whenever it is defined, is defined on (at least) all models in $[\![SP]\!]$,[5] and yields a result in $[\![SP']\!]$ when applied to a model in $[\![SP]\!]$.

There is a crucial difference here between monolithic self-contained programs and software components: while monolithic programs are modelled as CASL models, software components are modelled as (possibly partial) functions mapping CASL models of the parameter specification SP to CASL models of the result specification SP'.

The next important idea is that when assembling components, in general a given component will not be applied to a sub-system providing *exactly* what is required by the component; it will be applied to a sub-system providing *at least, and in general more than* what is required.

[4] Composition of derived signature morphisms can be defined in the evident fashion, and equality of two derived signature morphisms is understood here semantically.

[5] Intuitively, $\Delta P(P)$ is "statically" well-formed as soon as P has the right signature, but needs to be defined only for arguments that realize SP.

Technically, this means that we need to look at constructions that map Σ-models to Σ'-models, but applied to parts cut out of "larger" Σ_G-models, where this "cutting out" is given as the reduct with respect to a signature morphism $\gamma : \Sigma \to \Sigma_G$ that fits the local argument signature into its global context.

Throughout the rest of the paper, we will repeatedly refer to the signatures and morphisms in the following pushout diagram:

$$\begin{tikzcd} \Sigma_G \arrow[r, "\iota'"] & \Sigma'_G \\ \Sigma \arrow[u, "\gamma"] \arrow[r, "\iota"'] & \Sigma' \arrow[u, "\gamma'"'] \end{tikzcd}$$

where the local construction is along the bottom of the diagram, "cutting out" its argument from a larger model uses the signature morphism on the left, and the resulting global construction is along the top.

Definition 3.3 (Admissibility and global construction). Given a local construction F along a signature morphism $\iota : \Sigma \to \Sigma'$, a morphism $\gamma : \Sigma \to \Sigma_G$ fitting Σ into a "global" signature Σ_G is *admissible* if the pushout of ι and γ above ensures amalgamability. Then, for any Σ_G-model $\mathcal{G} \in |\mathbf{Mod}(\Sigma_G)|$, we define the *global result* $F_G(\mathcal{G})$ of applying F to $\mathcal{G}$ by reference to the pushout diagram above, using the amalgamation property: if $\mathcal{G}|_\gamma \in dom(F)$ then $F_G(\mathcal{G}) = \mathcal{G} \oplus F(\mathcal{G}|_\gamma)$; otherwise $F_G(\mathcal{G})$ is undefined.

This determines a *global construction* $F_G : |\mathbf{Mod}(\Sigma_G)| \rightharpoonup |\mathbf{Mod}(\Sigma'_G)|$, which is persistent along $\iota' : \Sigma_G \to \Sigma'_G$.

This way of "lifting" a persistent function to a larger context via a "*fitting morphism*" using signature pushout and amalgamation is well established in the algebraic specification tradition, going back at least to "parametrized specifications" with free functor semantics, see (Ehrig and Mahr 1985). The extra requirement here is that only admissible fitting morphisms are permitted, turning amalgamability into a (static) requirement for correct application of a local construction in a given context, to be discharged using the machinery of (Schröder *et al.* 2001; Klin *et al.* 2001; Schröder *et al.* 2005).

Then an obvious issue is whether a software component that realizes a component specification $SP \xrightarrow{\iota} SP'$, when combined with a sub-system that realizes a specification SP_G, actually provides a system that realizes a given specification SP'_G. The corresponding correctness condition is provided by the following theorem.

Theorem 3.4. Given a local construction $F \in [\![SP \xrightarrow{\iota} SP']\!]$, a specification SP_G with admissible fitting morphism $\gamma : Sig[SP] \to Sig[SP_G]$, and a specification SP'_G with $Sig[SP'_G] = \Sigma'_G$, the induced global construction F_G along $\iota' : \Sigma_G \to \Sigma'_G$ is literally correct w.r.t. SP_G and SP'_G, i.e., $F_G \in [\![SP_G \xrightarrow{\iota'} SP'_G]\!]$, provided that:

— $[\![SP_G]\!] \subseteq [\![SP \textbf{ with } \gamma]\!]$, and

— $[\![(SP' \textbf{ with } \gamma') \textbf{ and } (SP_G \textbf{ with } \iota')]\!] \subseteq [\![SP'_G]\!]$.

Proof. Let $\mathcal{G} \in [\![SP_G]\!]$. Then $\mathcal{G}|_\gamma \in [\![SP]\!]$, and so $\mathcal{G}|_\gamma \in dom(F)$ and $F(\mathcal{G}|_\gamma) \in [\![SP']\!]$. Consequently $F_G(\mathcal{G}) \in [\![SP' \textbf{ with } \gamma']\!] \cap [\![SP_G \textbf{ with } \iota']\!]$. □

Informally, this captures directly a "bottom-up" process of building component-based systems, whereby we start with SP_G, a specification of a "global" assembly of components built so far, find a local construction (a component) $F \in [\![SP \xrightarrow{\iota} SP']\!]$ with a fitting morphism γ that satisfies the first condition, and define SP'_G such that the second condition is satisfied (e.g. by simply taking $SP'_G = (SP'$ **with** $\gamma')$ **and** $(SP_G$ **with** $\iota'))$, thus obtaining a specification of the global assembly of components with the new component built using F added. When proceeding "top-down", we start with the global requirements specification SP'_G. To use a local construction (a component) $F \in [\![SP \xrightarrow{\iota} SP']\!]$, we have to decide which part of the requirements it is going to implement by providing a signature morphism $\gamma' \colon \mathit{Sig}[SP'] \to \mathit{Sig}[SP'_G]$, then construct the "pushout complement" $\gamma \colon \mathit{Sig}[SP] \to \Sigma_G$, $\iota' \colon \Sigma_G \to \mathit{Sig}[SP'_G]$ for ι and γ', and finally devise a specification SP_G with $\mathit{Sig}[SP_G] = \Sigma_G$ such that both conditions are satisfied. Then SP_G is the requirements specification for the components that remain to be implemented.

4. Architectural Specifications

Using local constructions for global implementations of specifications, we have moved only one step away from a monolithic global view of specifications and constructions used to implement them. The notion of *architectural specification* (Bidoit, Sannella and Tarlecki 2002a) as introduced for CASL takes us much further. An architectural specification *prescribes* a decomposition of the task of implementing a requirements specification into a number of subtasks to implement specifications of "modular components" (called *units*) of the system under development. The units may be parametrized, and then we can identify them with local constructions; non-parametrized units are just models. Another essential part of an architectural specification is a prescription of how the units, once developed, are to be put together using a few simple operators. One of these is application of a parametrized unit which corresponds exactly to the lifting of a local construction to a larger context studied above. Thus, an architectural specification may be thought of as a definition of a complex construction to be used in a top-down development process to implement a requirements specification by a number of specifications (of non-parametrized units), where the construction uses a number of specified local constructions that are to be developed as well.

For the sake of readability, we will discuss here a simplified version of CASL architectural specifications, with a limited (but representative) number of constructs, shaped after a version used in (Schröder *et al.* 2001; Schröder *et al.* 2005); a generalization to full architectural specifications (including unit renaming, units with multiple parameters, local unit definitions, etc.) would be tedious but rather straightforward, except perhaps for the "unguarded import" mechanism, see (Hoffman 2001). Our version of architectural specifications is defined as follows.

Architectural specifications: ASP ::= **arch spec** UDD^+ **result** T;

UDD ::= $\mathit{Dcl} \mid \mathit{Dfn}$

An architectural specification consists of a (non-empty) list of unit declarations or definitions followed by a unit result term.

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

A unit declaration introduces a unit name with its type, which is either a specification or a specification of a parametrized unit, determined by a specification of its parameter and its result that extends the parameter via a signature morphism ι.

Unit definitions: $Dfn ::= U = T$

A unit definition introduces a (non-parametrized) unit and gives its value by a unit term.

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

A unit term is either a (non-parametrized) unit name, or a unit restricted w.r.t. a signature morphism, or a unit application with an argument that fits via a signature morphism γ, or an amalgamation of units.

Following the semantics of full CASL (Baumeister *et al.* 2004), see also (Schröder *et al.* 2001; Schröder *et al.* 2005), we give the semantics of this CASL fragment in two stages: first we give its *extended static semantics*[6] and then its *literal model semantics*. (We refer to this as the *literal* model semantics by contrast with the *observational* model semantics of Sect. 7 below.)

For the extended static semantics we need a concept of *static context*, which carries signatures for the units declared or defined within an architectural specification, together with information on their mutual dependencies. Analogously, for the model semantics we need a concept of *environment*, which carries the semantics of the units named in the corresponding static context.

When discussing application of local constructions to global models in Sect. 3, we viewed the global context as a single monolithic model over a single "global" signature. Unfortunately, in the context of architectural specifications in CASL this view cannot be maintained. The technical reason is that CASL does not ensure amalgamation over arbitrary colimits of signature diagrams, as pointed out in Sect. 2. Indeed, if amalgamability were ensured for arbitrary colimits of signature diagrams, we could always represent the global context of all the (non-parametrized) units declared or defined so far by a monolithic global model over a single global signature, and many of the technicalities below become rather simpler, see (Bidoit, Sannella and Tarlecki 2002b; Tarlecki 2003). As things are, for architectural specifications in CASL, static information about (non-parametrized) units declared or defined in an architectural specification will be stored in signature diagrams, with nodes labeled by unit signatures and edges labeled by signature morphisms that capture dependencies between units.

More formally, we view a signature diagram as a graph morphism from its *shape* $\mathbf{I}$ to the category of CASL signatures, $D : \mathbf{I} \to \mathbf{Sig}$. We write $|D|$ for the set of nodes of $\mathbf{I}$, and $m{:}\, i \to j$ in D for an edge m with source i and target j in $\mathbf{I}$. The extension of diagrams is understood as usual. Two diagrams D_1, D_2 *disjointly extend* D if both D_1 and D_2 extend D and the intersection of their shapes is the shape of D. If this is the case then the union $D_1 \cup D_2$ is well-defined. As usual, disjointness of diagram extensions may be ensured by choosing the new nodes and edges appropriately.

[6] In (Baumeister *et al.* 2004), a distinction is made between *static semantics* of architectural specifications, which ignores dependencies between terms and hence does not contribute to the analysis of amalgamability, and *extended static semantics*.

For any diagram $D : \mathbf{I} \to \mathbf{Sig}$, a family $\mathcal{M} = \langle M_i \rangle_{i \in |D|}$ of models is called *D-coherent* if for each $i \in |D|$, $M_i \in |\mathbf{Mod}(D(i))|$, and for each $m: i \to j$ in $\mathbf{I}$, $M_i = M_j\big|_{D(m)}$; this is extended to $|D|$-indexed families of model morphisms in the obvious way. Given a D-coherent family $\mathcal{M} = \langle M_i \rangle_{i \in |D|}$, we write $\mathcal{M}_i$ for M_i, $i \in |D|$. We let $\mathbf{Mod}(D)$ be the category with D-coherent model families as objects and D-coherent families of model morphisms as morphisms (with the obvious component-wise composition).

D ensures amalgamability for D', where D' extends D, if any D-coherent model family can be uniquely extended to a D'-coherent model family. It is easy to see that Def. 2.1 is in fact a special case of this notion.[7]

An *extended static context* $\mathcal{C}_{st} = (P_{st}, \mathcal{B}_{st}, D)$, in which CASL phrases are elaborated, consists of a static context for parametrized units P_{st} mapping parametrized unit names to signature morphisms (from the parameter to the result signature), a global context diagram D, and a static context for non-parametrized units $\mathcal{B}_{st}$ mapping non-parametrized unit names to nodes in D. From any such extended static context we can extract a *static context* $ctx(\mathcal{C}_{st}) = (P_{st}, B_{st})$ by preserving the static context P_{st} for parametrized units and building a direct static context B_{st} for non-parametrized units that extracts their signatures from $\mathcal{B}_{st}$ and D (i.e., $B_{st}(U) = D_{\mathcal{B}_{st}(U)}$). $\mathcal{C}^{\emptyset}_{st}$ stands for the "empty" extended static context that consists of the empty parametrized and non-parametrized unit contexts, and of the empty context diagram. Extension (or inclusion) of extended static contexts, written $\mathcal{C}_{st} \subseteq \mathcal{C}'_{st}$, is defined component-wise, as expected. We refer to unit names in $dom(P_{st})$ as parametrized unit names in $\mathcal{C}_{st}$, and to those in $dom(\mathcal{B}_{st})$ as non-parameterized unit names in $\mathcal{C}_{st}$.

Figure 1 gives rules to derive semantic judgments of the following forms:

— $\vdash ASP \rhd\!\!\rhd ((P_{st}, B_{st}), \Sigma)$: the architectural specification ASP yields a static context describing the units declared or defined in ASP, and the signature of the result unit;

— $\vdash UDD^{+} \rhd\!\!\rhd \mathcal{C}_{st}$: the sequence UDD^{+} of unit declarations and definitions yields an extended static context $\mathcal{C}_{st}$;

— $\mathcal{C}_{st} \vdash UDD \rhd\!\!\rhd \mathcal{C}'_{st}$: the unit declaration or definition UDD in the extended static context $\mathcal{C}_{st}$ yields a new extended static context $\mathcal{C}'_{st}$ which extends $\mathcal{C}_{st}$;

— $(P_{st}, \mathcal{B}_{st}, D) \vdash T \rhd\!\!\rhd (i, D')$: the unit term T in the extended static context $(P_{st}, \mathcal{B}_{st}, D)$ yields a new context diagram D' which extends D and a node i in D' that carries the signature of the unit term T.

To follow the rules for unit application and amalgamation, it may be helpful to look at Fig. 2, where the corresponding global context diagrams are sketched.

It is worth noting that in the rule for parametrized unit application, the requirement that D' ensures amalgamability for D'' is weaker than requiring that the pushout used in this rule ensures amalgamability: even if it does not, the global context in which the

[7] In spite of Lemma 2.2 and its obvious generalization to colimits of arbitrary signature diagrams, we do not know whether in the framework of CASL it is always the case that if D ensures amalgamability for D' then also the similar property holds for D-coherent families of model morphisms; we conjecture that this is the case. However, in this paper we need only a few special cases of this, where D' arises from D essentially by adding a surjective cone, and so a proof similar to that for Lemma 2.2 goes through; the same is true for a similar generalisation of Lemma 5.6 below.

$$\frac{\vdash UDD^{+} \rhd\!\!\!\rhd \mathcal{C}_{st} \qquad \mathcal{C}_{st} \vdash T \rhd\!\!\!\rhd (i, D)}{\vdash \textbf{arch spec } UDD^{+} \textbf{ result } T \rhd\!\!\!\rhd (ctx(\mathcal{C}_{st}), D(i))}$$

$$\frac{\begin{array}{c}\mathcal{C}^{\emptyset}_{st} \vdash UDD_1 \rhd\!\!\!\rhd (\mathcal{C}_{st})_1 \\ \cdots \\ (\mathcal{C}_{st})_{n-1} \vdash UDD_n \rhd\!\!\!\rhd (\mathcal{C}_{st})_n\end{array}}{\vdash UDD_1 \ldots UDD_n \rhd\!\!\!\rhd (\mathcal{C}_{st})_n}$$

$$\frac{\begin{array}{c}U \notin (dom(P_{st}) \cup dom(\mathcal{B}_{st})) \\ D' \text{ extends } D \text{ by a new node } i \text{ with } D'(i) = Sig(SP)\end{array}}{(P_{st}, \mathcal{B}_{st}, D) \vdash U{:}\, SP \rhd\!\!\!\rhd (P_{st}, \mathcal{B}_{st} + \{U \mapsto i\}, D')}$$

$$\frac{\begin{array}{c}\iota : Sig(SP_1) \to Sig(SP_2) \\ U \notin (dom(P_{st}) \cup dom(\mathcal{B}_{st}))\end{array}}{(P_{st}, \mathcal{B}_{st}, D) \vdash U{:}\, SP_1 \xrightarrow{\iota} SP_2 \rhd\!\!\!\rhd (P_{st} + \{U \mapsto \iota\}, \mathcal{B}_{st}, D)}$$

$$\frac{(P_{st}, \mathcal{B}_{st}, D) \vdash T \rhd\!\!\!\rhd (i, D') \qquad U \notin (dom(P_{st}) \cup dom(\mathcal{B}_{st}))}{(P_{st}, \mathcal{B}_{st}, D) \vdash U = T \rhd\!\!\!\rhd (P_{st}, \mathcal{B}_{st} + \{U \mapsto i\}, D')}$$

$$\frac{U \in dom(\mathcal{B}_{st})}{(P_{st}, \mathcal{B}_{st}, D) \vdash U \rhd\!\!\!\rhd (\mathcal{B}_{st}(U), D)}$$

$$\frac{\begin{array}{c}\mathcal{C}_{st} \vdash T \rhd\!\!\!\rhd (i, D) \qquad \sigma{:}\, \Sigma \to D(i) \\ D' \text{ extends } D \text{ by a new node } j \text{ and a new edge } m{:}\, j \to i \text{ with } D'(m) = \sigma\end{array}}{\mathcal{C}_{st} \vdash \textbf{reduce } T \textbf{ by } \sigma \rhd\!\!\!\rhd (j, D')}$$

$$\frac{\begin{array}{c}P_{st}(U) = \iota{:}\, \Sigma \to \Sigma' \qquad \mathcal{C}_{st} \vdash T \rhd\!\!\!\rhd (i, D) \qquad \gamma{:}\, \Sigma \to D(i) \\ (\iota', \gamma') \text{ is a pushout of } (\gamma, \iota) \\ D' \text{ extends } D \text{ by new nodes } j, k \text{ and edges } m{:}\, j \to i, n{:}\, j \to k \\ \qquad\qquad\qquad \text{with } D'(m) = \gamma, D'(n) = \iota \\ D'' \text{ extends } D' \text{ by a new node } l \text{ and edges } r{:}\, i \to l, s{:}\, k \to l \\ \qquad\qquad\qquad \text{with } D''(r) = \iota', D''(s) = \gamma' \\ D' \text{ ensures amalgamability for } D''\end{array}}{\mathcal{C}_{st} \vdash U[T \textbf{ fit } \gamma] \rhd\!\!\!\rhd (l, D'')}$$

$$\frac{\begin{array}{c}(P_{st}, \mathcal{B}_{st}, D) \vdash T_1 \rhd\!\!\!\rhd (i_1, D_1) \qquad (P_{st}, \mathcal{B}_{st}, D) \vdash T_2 \rhd\!\!\!\rhd (i_2, D_2) \\ D_1 \text{ and } D_2 \text{ are disjoint extensions of } D \\ D' \text{ extends } D_1 \cup D_2 \text{ by a new node } j \text{ and edges } m_1{:}\, i_1 \to j, m_2{:}\, i_2 \to j \text{ with} \\ \qquad D'(j) = D_1(i_1) \cup D_2(i_2), D'(m_1){:}\, D_1(i_1) \hookrightarrow D'(j), D'(m_2){:}\, D_2(i_2) \hookrightarrow D'(j) \\ D_1 \cup D_2 \text{ ensures amalgamability for } D'\end{array}}{(P_{st}, \mathcal{B}_{st}, D) \vdash T_1 \textbf{ and } T_2 \rhd\!\!\!\rhd (j, D')}$$

Fig. 1. Extended static semantics

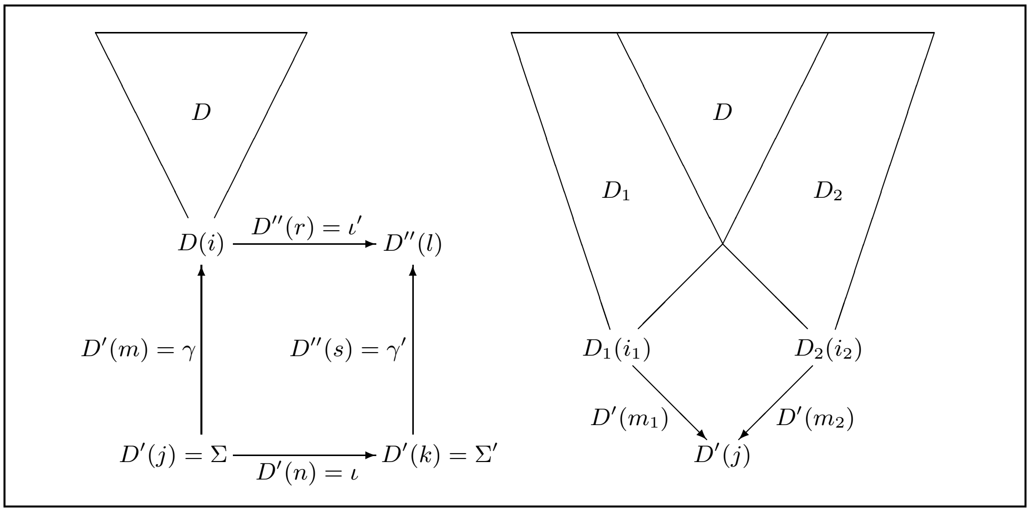

Fig. 2. Unit application and amalgamation diagrams

application is carried out may impose additional constraints on the models involved that ensure amalgamability.

Note also that the rule for unit amalgamation does not require that the amalgamated units have common signatures: the resulting unit will be built over the union of the two signatures, provided that this union is defined[8] and that the two units built can be uniquely amalgamated to yield a unit over this union signature. This is ensured by the final condition in the rule, which requires that the dependencies between units captured in the diagram $D_1 \cup D_2$ ensure amalgamability of the two models involved. This requires in particular that these models share the interpretation of the symbols in the intersection of their signatures.

In the model semantics we work with *contexts* $\mathcal{C}$ that are classes of *unit environments* E. Unit environments map unit names to either local constructions (for parametrized units) or to individual models (for non-parametrized units). *Unit evaluators* UEv map unit environments to models.

Given an extended static context $\mathcal{C}_{st} = (P_{st}, \mathcal{B}_{st}, D)$, a unit environment E *fits* $\mathcal{C}_{st}$ if

— for each $U \in dom(P_{st})$, $E(U)$ is a local construction along $P_{st}(U)$, and
— there is a D-coherent family of models $\mathcal{M} \in |\mathbf{Mod}(D)|$ such that for each $U \in dom(\mathcal{B}_{st})$, $E(U) = \mathcal{M}_{\mathcal{B}_{st}(U)}$; we say then that $\mathcal{M}$ *witnesses* E *in* $\mathcal{C}_{st}$.

We write $ucx(\mathcal{C}_{st})$ for the class of all unit environments that fit $\mathcal{C}_{st}$. Note that if $\mathcal{C}_{st} \subseteq \mathcal{C}'_{st}$ then $ucx(\mathcal{C}'_{st}) \subseteq ucx(\mathcal{C}_{st})$.

[8] The union is defined in the obvious, component-wise manner, with the subsort preorder given as the transitive closure of the two preorders in the component signatures — however, this may fail to yield a Casl signature due to overloading of operation and predicate names that may arise, which we have disallowed here. The union may also fail to be defined with Casl's treatment of overloading, albeit for more subtle reasons.

Two unit environments $E_1, E_2 \in ucx(\mathcal{C}_{st})$ *coincide in* $\mathcal{C}_{st}$, written $E_1 =_{\mathcal{C}_{st}} E_2$, if for all (parametrized and non-parametrized) unit names U in $\mathcal{C}_{st}$, $E_1(U) = E_2(U)$.

Proposition 4.1. If $E_1 =_{\mathcal{C}_{st}} E_2$ then any family that witnesses E_1 in $\mathcal{C}_{st}$, witnesses E_2 in $\mathcal{C}_{st}$ as well. $\square$

A context $\mathcal{C} \subseteq ucx(\mathcal{C}_{st})$ is *closed in* $\mathcal{C}_{st}$ if for all unit environments $E_1 \in \mathcal{C}$ and $E_2 \in ucx(\mathcal{C}_{st})$, $E_1 =_{\mathcal{C}_{st}} E_2$ implies $E_2 \in \mathcal{C}$.

$\mathcal{C}^{\emptyset} = ucx(\mathcal{C}^{\emptyset}_{st})$ is the context which constrains no unit name. Given a context $\mathcal{C}$, a unit name U and a class of units $\mathcal{V}$, we write $\mathcal{C} \times \{U \mapsto \mathcal{V}\}$ for $\{E + \{U \mapsto V\} \mid E \in \mathcal{C}, V \in \mathcal{V}\}$, where $E + \{U \mapsto V\}$ maps U to V and otherwise behaves like E.

Figure 3 gives rules to derive semantic judgments of the following forms:

— $\vdash ASP \Rightarrow (\mathcal{C}, UEv)$: the architectural specification ASP yields a context $\mathcal{C}$ with environments providing interpretations for the units declared and defined in ASP, and a unit evaluator that for each such environment determines the result unit;
— $\vdash UDD^+ \Rightarrow \mathcal{C}$: the sequence UDD^+ of unit declarations and definitions yields a context $\mathcal{C}$;
— $\mathcal{C} \vdash UDD \Rightarrow \mathcal{C}'$: the unit declaration or definition UDD in the context $\mathcal{C}$ yields a new context $\mathcal{C}'$;
— $\mathcal{C} \vdash T \Rightarrow UEv$: the unit term T in the context $\mathcal{C}$ yields a unit evaluator UEv that when given an environment (in $\mathcal{C}$) yields the unit resulting from the evaluation of T in this environment.

The rules rely on a successful run of the extended static semantics; this allows us to use the static concepts and notations introduced there. The crossed-out premises in the rules are crucial properties that are guaranteed to hold for phrases for which the extended static semantics yields a result, as a consequence of the following theorem.

Theorem 4.2. The following invariants link the extended static semantics and model semantics:

1 If $\vdash ASP \rhd\!\!\rhd ((P_{st}, B_{st}), \Sigma)$ and $\vdash ASP \Rightarrow (\mathcal{C}, UEv)$ then there is an extended static context $\mathcal{C}_{st}$ such that $ctx(\mathcal{C}_{st}) = (P_{st}, B_{st})$ and $\mathcal{C} \subseteq ucx(\mathcal{C}_{st})$, $\mathcal{C}$ is closed in $\mathcal{C}_{st}$, and for each $E \in \mathcal{C}$, $E \in dom(UEv)$ and $UEv(E) \in |\mathbf{Mod}(\Sigma)|$. Moreover, for $E_1, E_2 \in \mathcal{C}$, if $E_1 =_{\mathcal{C}_{st}} E_2$ then $UEv(E_1) = UEv(E_2)$.
2 If $\vdash UDD^+ \rhd\!\!\rhd \mathcal{C}_{st}$ and $\vdash UDD^+ \Rightarrow \mathcal{C}$ then $\mathcal{C} \subseteq ucx(\mathcal{C}_{st})$ and $\mathcal{C}$ is closed in $\mathcal{C}_{st}$.
3 If $\mathcal{C}_{st} \vdash UDD \rhd\!\!\rhd \mathcal{C}'_{st}$ and $\mathcal{C} \vdash UDD \Rightarrow \mathcal{C}'$, where $\mathcal{C} \subseteq ucx(\mathcal{C}_{st})$ and $\mathcal{C}$ is closed in $\mathcal{C}_{st}$, then $\mathcal{C}' \subseteq ucx(\mathcal{C}'_{st})$, $\mathcal{C}' \subseteq \mathcal{C}$, $\mathcal{C}'$ is closed in $\mathcal{C}'_{st}$ and for each unit environment $E \in \mathcal{C}$ and model family $\mathcal{M}$ that witnesses E in $\mathcal{C}_{st}$, there is $E' \in \mathcal{C}'$ such that $E =_{\mathcal{C}_{st}} E'$ and an extension of $\mathcal{M}$ witnesses E' in $\mathcal{C}'_{st}$.
4 If $\mathcal{C}_{st} \vdash T \rhd\!\!\rhd (i, D')$ and $\mathcal{C} \vdash T \Rightarrow UEv$ with $\mathcal{C} \subseteq ucx(\mathcal{C}_{st})$, then for each unit environment $E \in \mathcal{C}$ and model family $\mathcal{M}$ that witnesses E in $\mathcal{C}_{st}$, there is an extension of $\mathcal{M}$ to a D'-coherent model family $\mathcal{M}' \in |\mathbf{Mod}(D')|$ such that $\mathcal{M}'_i = UEv(E)$. Moreover, for $E_1, E_2 \in \mathcal{C}$, if $E_1 =_{\mathcal{C}_{st}} E_2$ then $UEv(E_1) = UEv(E_2)$.

Proof. Item 4 is proved by induction on the structure of the unit term. The fact that the value of the unit evaluator on an environment does not change when it does not depend

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

$$\frac{\begin{array}{c}\mathcal{C}^{\emptyset} \vdash UDD_1 \Rightarrow \mathcal{C}_1 \\ \cdots \\ \mathcal{C}_{n-1} \vdash UDD_n \Rightarrow \mathcal{C}_n\end{array}}{\vdash UDD_1 \ldots UDD_n \Rightarrow \mathcal{C}_n}$$

$$\frac{}{\mathcal{C} \vdash U \colon SP \Rightarrow \mathcal{C} \times \{U \mapsto [\![SP]\!]\}}$$

$$\frac{}{\mathcal{C} \vdash U \colon SP_1 \xrightarrow{\iota} SP_2 \Rightarrow \mathcal{C} \times \{U \mapsto [\![SP_1 \xrightarrow{\iota} SP_2]\!]\}}$$

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

$$\frac{}{\mathcal{C} \vdash U \Rightarrow \lambda E \in \mathcal{C} \cdot E(U)}$$

$$\frac{\mathcal{C} \vdash T \Rightarrow UEv}{\mathcal{C} \vdash \textbf{reduce } T \textbf{ by } \sigma \Rightarrow \lambda E \in \mathcal{C} \cdot UEv(E)|_{\sigma}}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash T \Rightarrow UEv \qquad \text{for each } E \in \mathcal{C}, UEv(E)|_{\gamma} \in dom(E(U)) \\ \cancel{\text{for each } E \in \mathcal{C}, UEv(E) \oplus E(U)(UEv(E)|_{\gamma}) \text{ is well-defined}}\end{array}}{\mathcal{C} \vdash U[T \textbf{ fit } \gamma] \Rightarrow \lambda E \in \mathcal{C} \cdot UEv(E) \oplus E(U)(UEv(E)|_{\gamma})}$$

$$\frac{\begin{array}{c}\mathcal{C} \vdash T_1 \Rightarrow UEv_1 \qquad \mathcal{C} \vdash T_2 \Rightarrow UEv_2 \\ \cancel{\text{for each } E \in \mathcal{C}, \text{ there is a unique } M \in |\mathbf{Mod}(\Sigma)| \text{ such that}} \\ \cancel{M|_{\iota_1} = UEv_1(E), M|_{\iota_2} = UEv_2(E)} \\ UEv = \{E \mapsto M \mid E \in \mathcal{C}, M|_{\iota_1} = UEv_1(E), M|_{\iota_2} = UEv_2(E)\}\end{array}}{\mathcal{C} \vdash T_1 \textbf{ and } T_2 \Rightarrow UEv}$$

Fig. 3. Literal model semantics

on the values in the environment not mentioned in the static context (for $E_1, E_2 \in \mathcal{C}$, if $E_1 =_{\mathcal{C}_{st}} E_2$ then $UEv(E_1) = UEv(E_2)$) follows in each case easily, using the inductive hypothesis.

The case of unit name is trivial, and the case of unit reduct is very easy.

Consider the case of unit application, when the unit term is of the form $U[T \textbf{ fit } \gamma]$. Adjusting the notation slightly to fit the corresponding rules (for unit application) in Figs. 1 and 3 (we will implicitly rely below on the notation used in these rules), assume that $\mathcal{C} \subseteq ucx(\mathcal{C}_{st})$, $\mathcal{C}_{st} \vdash U[T \textbf{ fit } \gamma] \rhd\!\!\rhd (l, D'')$ and $\mathcal{C} \vdash U[T \textbf{ fit } \gamma] \Rightarrow UEv'$, where $UEv'(E) = UEv(E) \oplus E(U)(UEv(E)|_{\gamma})$ for $E \in \mathcal{C}$. Consequently, all the premises of the corresponding rules (for unit application) in Figs. 1 and 3 must hold. Let $E \in \mathcal{C}$ and $\mathcal{M}$ be a model family that witnesses E in $\mathcal{C}_{st}$. By the inductive hypothesis, there is an extension $\mathcal{M}^T \in |\mathbf{Mod}(D)|$ of $\mathcal{M}$ such that $\mathcal{M}^T_i = UEv(E)$. Let $\mathcal{M}'$ extend $\mathcal{M}^T$ by putting $\mathcal{M}'_j = UEv(E)\big|_\gamma$ and $\mathcal{M}'_k = E(U)(UEv(E)\big|_\gamma)$ (since $UEv(E)\big|_\gamma \in dom(E(U))$, the latter is well-defined). Then $\mathcal{M}' \in |\mathbf{Mod}(D')|$. Since D' ensures amalgamability for D'', $\mathcal{M}'$ uniquely extends to $\mathcal{M}'' \in |\mathbf{Mod}(D'')|$, yielding $\mathcal{M}''_l\big|_{\iota'} = \mathcal{M}'_i$ and $\mathcal{M}''_l\big|_{\gamma'} = \mathcal{M}'_k$, that is, $\mathcal{M}''_l = UEv(E) \oplus E(U)(UEv(E)\big|_\gamma)$ — which completes the proof for this case.

For the case of unit amalgamation, when the unit term is of the form T_1 **and** T_2, assume $\mathcal{C} \subseteq ucx(\mathcal{C}_{st})$, $\mathcal{C}_{st} \vdash T_1$ **and** $T_2 \rhd\!\!\rhd (j, D')$ and $\mathcal{C} \vdash T_1$ **and** $T_2 \Rightarrow UEv$, where $\mathcal{C}_{st} = (P_{st}, \mathcal{B}_{st}, D)$. Consequently, all the premises of the corresponding rules (for unit amalgamation) in Figs. 1 and 3 must hold; we refer below to the notations used in the rules. Let $E \in \mathcal{C}$ and $\mathcal{M}$ be a model family that witnesses E in $\mathcal{C}_{st}$. By the inductive hypothesis, there are extensions $\mathcal{M}^1 \in |\mathbf{Mod}(D_1)|$ and $\mathcal{M}^2 \in |\mathbf{Mod}(D_2)|$ of $\mathcal{M}$ such that $\mathcal{M}^1_{i_1} = UEv_1(E)$ and $\mathcal{M}^2_{i_2} = UEv_2(E)$. Since D_1 and D_2 are disjoint extensions of D, $\mathcal{M}^1 \cup \mathcal{M}^2$ is a $(D_1 \cup D_2)$-coherent family of models. Now, since $D_1 \cup D_2$ ensures amalgamability for D', $\mathcal{M}^1 \cup \mathcal{M}^2$ extends uniquely to a D'-coherent family $\mathcal{M}' \in |\mathbf{Mod}(D')|$, necessarily with $\mathcal{M}'_j\big|_{D'(m_1)} = \mathcal{M}^1_{i_1}$ and $\mathcal{M}'_j\big|_{D'(m_2)} = \mathcal{M}^2_{i_2}$, that is, $\mathcal{M}'_j = UEv(E)$ — which completes the proof of item 4.

Item 3 follows by inspection of the rules; the cases of unit declarations are easy. The case of unit definitions relies on item 4 as follows. Assume that $\mathcal{C} \subseteq ucx(\mathcal{C}_{st})$ and $\mathcal{C}$ is closed in $\mathcal{C}_{st}$, $\mathcal{C}_{st} = (P_{st}, \mathcal{B}_{st}, D)$. To derive $\mathcal{C}_{st} \vdash UDD \rhd\!\!\rhd \mathcal{C}'_{st}$ and $\mathcal{C} \vdash UDD \Rightarrow \mathcal{C}'$, where UDD is of the form $U = T$, we must have $(P_{st}, \mathcal{B}_{st}, D) \vdash T \rhd\!\!\rhd (i, D')$, $U \notin (dom(P_{st}) \cup dom(\mathcal{B}_{st}))$, and $\mathcal{C} \vdash T \Rightarrow UEv$, with $\mathcal{C}'_{st} = (P_{st}, \mathcal{B}_{st} + \{U \mapsto i\}, D')$ and $\mathcal{C}' = \{E + \{U \mapsto UEv(E)\} \mid E \in \mathcal{C}\}$. Now, for each $E \in \mathcal{C}$ and model family $\mathcal{M} \in |\mathbf{Mod}(D)|$ that witnesses E in $\mathcal{C}_{st}$, by item 4 there exists an extension $\mathcal{M}' \in |\mathbf{Mod}(D')|$ of $\mathcal{M}$ with $\mathcal{M}'_i = UEv(E)$. $\mathcal{M}'$ witnesses $E + \{U \mapsto UEv(E)\}$ in $\mathcal{C}'_{st}$. Consequently, we have $\mathcal{C}' \subseteq ucx(\mathcal{C}'_{st})$. Moreover, since $\mathcal{C}$ is closed in $\mathcal{C}_{st}$ and $U \notin (dom(P_{st}) \cup dom(\mathcal{B}_{st}))$, $(E + \{U \mapsto UEv(E)) \in \mathcal{C}$, which shows $\mathcal{C}' \subseteq \mathcal{C}$. Finally, $\mathcal{C}'$ is closed in $\mathcal{C}'_{st}$ since $\mathcal{C}$ is closed in $\mathcal{C}_{st}$.

Item 2 follows from item 3 by an obvious induction on the length of the sequence of unit declarations and definitions.

Finally, item 1 follows from items 2 and 4 by inspection of the rules. Namely, to derive the assumptions for ASP of the form **arch spec** UDD^+ **result** T, we must have $\vdash UDD^+ \rhd\!\!\rhd \mathcal{C}_{st}$ and $\vdash UDD^+ \Rightarrow \mathcal{C}$, as well as $\mathcal{C}_{st} \vdash T \rhd\!\!\rhd (i, D)$ and $\mathcal{C} \vdash T \Rightarrow UEv$, with $(P_{st}, B_{st}) = ctx(\mathcal{C}_{st})$ and $\Sigma = D(i)$. The thesis now follows directly from items 2 and 4. $\square$

The invariants in Thm. 4.2 ensure that the crossed out premises of the unit amalgamation rule and of the parametrized unit application rule in the literal model semantics follow from the other premises of the rule and the premises of the corresponding rules of the extended static semantics.

5. Observational Equivalence for CASL Models

So far, we have followed the usual interpretation for basic specifications given as sets of axioms over some signature, which is to require models of such a basic specification to satisfy all its axioms. This is what is captured by the notion of literal correctness

(Def. 3.2) and the literal model semantics of Fig. 3. However, in many practical examples this turns out to be overly restrictive. The point is that only a subset of the sorts in the signature of a specification are typically intended to be directly observable while the others are treated as internal, with properties of their elements made visible only via *observations*: terms producing a result of an observable sort, and predicates. Often there are models that do not satisfy the axioms "literally" but in which all observations nevertheless deliver the required results. This calls for a relaxation of the interpretation of specifications, as advocated in numerous "observational" or "behavioural" approaches, going back at least to (Giarratana, Gimona and Montanari 1976; Reichel 1981). Two general approaches are possible:

— introduce an "internal" *observational indistinguishability* relation between elements in the carrier of each model, and re-interpret equality in the axioms as indistinguishability; or

— introduce an "external" *observational equivalence* relation on models over each signature, and re-interpret specifications by closing their class of models under such equivalence.

It turns out that under some acceptable technical conditions, these two approaches are closely related and coincide for most basic specifications (Bidoit, Hennicker and Wirsing 1995; Bidoit and Tarlecki 1996). We follow the second approach here.

From now on we will assume that the set of observable sorts is empty and so predicates are the only observations. In view of this decision, there is no need to parametrize the definitions below by a chosen set of observable sorts. This departs from standard approaches to observational equivalence in the usual algebraic frameworks, where choosing a non-empty set of observable sorts is crucial to have any observations at all. Moreover, it is appropriate for this set to vary in the process of modular development, where some sorts must be locally considered as observable (e.g. the parameter sorts in specifications of local constructions). The former is taken care of by assuming that appropriate predicates are introduced into the specifications considered. For instance, to make a generated sort observable, it is enough to introduce the "equality predicate" on this sort into the specification.[9] The latter will be achieved in a technically different way here, see Def. 6.9 below and the subsequent comment.

We should also note here that for each CASL signature Σ and sort s in Σ we have $s \leq s$, and so we also have a predicate $in^{s \leq s}\colon s$, which holds for all its arguments in any CASL model. This means that given a Σ-term t of sort s, $in^{s \leq s}(t)$ holds if and only if t has a defined value. Consequently, observing predicates in CASL models covers observing definedness of terms.

Given a CASL signature Σ, an *observation* is an atomic predicate formula ϕ of the form $p(t_1, \ldots, t_n)$, where $p\colon s_1 \times \cdots \times s_n$ is a predicate symbol in $\Sigma^{\#}$ and for $i = 1, \ldots, n$, t_i is a

[9] Some free datatype definitions in CASL ensure that the new sort is observable even though no equality predicate is explicitly introduced. This is the case when there is a subsort for each alternative and selectors for each non-constant constructor. Then enough observations are available to distinguish between any two data values, provided that the other argument sorts for the constructors are observable (come with enough observations to distinguish between any data of these sorts).

$\Sigma^{\#}$-term of sort s_i. The observation $p(t_1, \ldots, t_n)$ is *closed* if all the terms t_i, $i = 1, \ldots, n$, are closed (contain no variables). Given a sort s in Σ, the observation $p(t_1, \ldots, t_n)$ is *for sort* s if it contains a unique variable $z{:}\, s$ of sort s (and no other variables at all). We will often write then $\phi(z)$ to indicate the variable explicitly, and for a $\Sigma^{\#}$-term t of sort s, we write $\phi(t)$ for the result of substituting t for z in ϕ.

Definition 5.1 (Observational equivalence). Given a CASL signature Σ, two Σ-models $M, N \in |\mathbf{Mod}(\Sigma)|$ are *observationally equivalent*, written $M \equiv N$, if for all closed observations ϕ,

$$M \models \phi \iff N \models \phi$$

It is trivial to see that observational equivalence is indeed an equivalence on CASL models over any signature Σ.

In the following we will work with a technically different but equivalent definition of observational equivalence, where the equivalence of two models is "witnessed" by a relation between them; this has been worked out in detail (for partial algebras without predicates) in (Schoett 1987), cf. "simulations" in (Milner 1971) and "weak homomorphisms" in (Ginzburg 1968).

Definition 5.2 (Correspondence). Consider a signature Σ. A *correspondence* between two Σ-models $M, N \in |\mathbf{Mod}(\Sigma)|$, written $\rho{:}\, M \bowtie N$, is a relation $\rho \subseteq |M| \times |N|$ that

— *is closed under the operations*: for $f{:}\, s_1 \times \ldots \times s_n \to s$ in $\Sigma^{\#}$, $a_1 \in |M|_{s_1}, \ldots, a_n \in |M|_{s_n}$ and $b_1 \in |N|_{s_1}, \ldots, b_n \in |N|_{s_n}$, if $(a_1, b_1) \in \rho_{s_1}, \ldots, (a_n, b_n) \in \rho_{s_n}$ then $f_M(a_1, \ldots, a_n)$ is defined iff $f_M(b_1, \ldots, b_n)$ is defined, and if so then $(f_M(a_1, \ldots, a_n), f_N(b_1, \ldots, b_n)) \in \rho_s$; and

— *preserves and reflects the predicates*: for $p{:}\, s_1 \times \ldots \times s_n$ in $\Sigma^{\#}$, $a_1 \in |M|_{s_1}, \ldots, a_n \in |M|_{s_n}$ and $b_1 \in |N|_{s_1}, \ldots, b_n \in |N|_{s_n}$, if $(a_1, b_1) \in \rho_{s_1}, \ldots, (a_n, b_n) \in \rho_{s_n}$ then $p_M(a_1, \ldots, a_n) \iff p_N(b_1, \ldots, b_n)$.

In the rest of the paper we will rely on the following equivalence without further mention:

Theorem 5.3. Given a CASL signature Σ, Σ-models $M, N \in |\mathbf{Mod}(\Sigma)|$ are observationally equivalent if and only if there is a correspondence between them.

Proof. Let $M \equiv N$. Define a relation $\rho \subseteq |M| \times |N|$ to contain, for each sort s in Σ, all and only pairs of the form (t_M, t_N), for all closed $\Sigma^{\#}$-terms t of sort s such that the value of t is defined in both M and N. To check that ρ is a correspondence between M and N, consider for $i = 1, \ldots, n$, $a_i \in |M|_{s_i}$ and $b_i \in |N|_{s_i}$ such that $(a_i, b_i) \in \rho_{s_i}$, so that $a_i = (t_i)_M$ and $b_i = (t_i)_N$ for some $\Sigma^{\#}$-term t_i of sort s_i. Consider now $f{:}\, s_1 \times \ldots \times s_n \to s$ in $\Sigma^{\#}$. Since $M \equiv N$, $M \models in^{s \leq s}(f(t_1, \ldots, t_n))$ iff $N \models in^{s \leq s}(f(t_1, \ldots, t_n))$; so, $f_M(a_1, \ldots, a_n)$ is defined iff $f_M(b_1, \ldots, b_n)$ is defined, and if so then by definition $(f_M(a_1, \ldots, a_n), f_N(b_1, \ldots, b_n)) \in \rho_s$ (since $(f(t_1, \ldots, t_n))_M = f_M(a_1, \ldots, a_n)$ and $(f(t_1, \ldots, t_n))_N = f_N(b_1, \ldots, b_n)$). Similarly, for $p{:}\, s_1 \times \ldots \times s_n$ in $\Sigma^{\#}$, $M \models p(t_1, \ldots, t_n)$ iff $N \models p(t_1, \ldots, t_n)$, which shows the equivalence of $p_M(a_1, \ldots, a_n)$ and $p_N(b_1, \ldots, b_n)$, and completes the proof of $\rho{:}\, M \bowtie N$.

Consider now a correspondence $\rho: M \bowtie N$. Using the correspondence properties, by simple induction on the term structure, for any closed $\Sigma^{\#}$-term t, one can prove that t_M is defined iff t_N is defined, and if so then $(t_M, t_N) \in \rho$. Now, given any closed observation $p(t_1, \ldots, t_n)$, by symmetry it is enough to prove that if $M \models p(t_1, \ldots, t_n)$ then also $N \models p(t_1, \ldots, t_n)$. Suppose $M \models p(t_1, \ldots, t_n)$. Then for $i = 1, \ldots, n$, $(t_i)_M$ is defined, and so $(t_i)_N$ is defined and $((t_i)_M, (t_i)_N) \in \rho$. Moreover, $p_M((t_1)_M, \ldots, (t_n)_M)$ holds, so by the correspondence property, $p_N((t_1)_N, \ldots, (t_n)_N)$ holds as well. Thus $N \models p(t_1, \ldots, t_n)$. $\square$

It is easy to check that isomorphisms (and in particular, identities) are correspondences and that the class of correspondences is closed under composition.

Correspondences between CASL models may be replaced by spans of strong homomorphisms. Namely, given a span of strong homomorphisms $(h_M: K \to M, h_N: K \to N)$, putting $\rho = h_M^{-1}; h_N$, i.e. $\rho_s = \{(h_M(c), h_N(c)) \mid c \in |K|_s\}$ for each sort s in Σ, yields a correspondence $\rho: M \bowtie N$. In the opposite direction:

Proposition 5.4. For any CASL signature Σ, Σ-models M, N and correspondence $\rho: M \bowtie N$, there is a Σ-model K and strong Σ-homomorphisms $h_M: K \to M$ and $h_N: K \to N$ such that $\rho = h_M^{-1}; h_N$.

Proof. To define K, first put $|K|_s = \rho_s \subseteq |M|_s \times |N|_s$ for each sort s in Σ. The operations in K are then defined component-wise, using the operations in M and N respectively. The predicates in K are defined either using the first components and the predicates in M, or (equivalently) using the second components and the predicates in N. The correspondence properties of ρ ensure that no problems arise, and that the projection functions $h_M: K \to M$ and $h_N: K \to N$ are strong Σ-homomorphisms. $\square$

This proposition directly implies that the reduct of a correspondence along a signature morphism (defined in the obvious way) is a correspondence. More interestingly, this extends to derived signature morphisms with observable conditions.

Consider a signature Σ. A conditional Σ-term $\langle(\phi_i, t_i)\rangle_{i\geq 0}$ is *observationally sensible* if for all $i \geq 0$, ϕ_i are *observers*, that is, Boolean combinations of observations. A derived signature morphism $\delta: \Sigma' \to \Sigma$ is *observationally sensible* if it maps Σ'-operations to observationally sensible terms.

Lemma 5.5. Let $\delta: \Sigma' \to \Sigma$ be an observationally sensible derived signature morphism, and let $\rho: M \bowtie N$ be a correspondence between Σ-models $M, N \in |\mathbf{Mod}(\Sigma)|$. Then $\rho|_\delta: M|_\delta \bowtie N|_\delta$ is a correspondence as well. $\square$

It follows that reducts with respect to observationally sensible derived signature morphisms extend to strong homomorphisms.

The view of correspondences as spans of homomorphisms also leads to an easy extension to correspondences of the amalgamation property given in Lemma 2.2 for homomorphisms:

Lemma 5.6. Suppose that the following pushout

$$\begin{tikzcd}
\Sigma_1 \arrow[r, "\iota'"] & \Sigma'_1 \\
\Sigma \arrow[u, "\gamma"] \arrow[r, "\iota"'] & \Sigma' \arrow[u, "\gamma'"']
\end{tikzcd}$$

ensures amalgamability. Then for all correspondences $\rho_1 \colon M_1 \bowtie N_1$ in $\mathbf{Mod}(\Sigma_1)$ and $\rho' \colon M' \bowtie N'$ in $\mathbf{Mod}(\Sigma')$ such that $\rho_1|_\gamma = \rho'|_\iota$ there exists a unique correspondence $\rho'_1 \colon M'_1 \bowtie N'_1$ in $\mathbf{Mod}(\Sigma'_1)$ such that $\rho'_1|_{\iota'} = \rho_1$ and $\rho'_1|_{\gamma'} = \rho'$, where $M'_1 = M_1 \oplus M'$ and $N'_1 = N_1 \oplus N'$.

Proof. A direct proof mimics the proof of Lemma 2.2. □

Note though that this does not ensure that amalgamation preserves observational equivalence:

Counterexample 5.7. Let Σ be a signature with a single sort s, and let Σ_1 extend Σ by a constant $a \colon s$. Since there are no predicates in Σ_1, all Σ_1-models where the constant a is defined are observationally equivalent. Let Σ' extend Σ by a unary predicate $p \colon s$; since there are no closed observations over Σ', all Σ'-models are observationally equivalent. The pushout signature of the two extensions of Σ is the signature Σ'_1 with sort s, constant $a \colon s$ and predicate $p \colon s$. Clearly, not all Σ'_1-models with defined values of a are observationally equivalent — there is a new closed observation here, namely $p(a)$.

To make the counterexample explicit, let M_1 be a Σ_1-model with a single element, $|M_1|_s = \{x\}$, and $a_{M_1} = x$. Let M' and M'' be Σ'-models such that $M'|_\Sigma = M''|_\Sigma = M_1|_\Sigma$ and $p_{M'}(x)$ holds while $p_{M''}(x)$ does not hold. Still, we have $M' \equiv M''$ (and trivially $M_1 \equiv M_1$). However, $(M_1 \oplus M') \not\equiv (M_1 \oplus M'')$. □

Observational equivalence can also be characterized in terms of internal indistinguishability. Namely, consider a CASL signature Σ and Σ-model $M \in |\mathbf{Mod}(\Sigma)|$. Let $\langle M \rangle$ be the generated submodel of M having all and only the defined values in M of closed $\Sigma^\#$-terms as elements of the carrier. For any sort s in Σ, given $a, a' \in |\langle M \rangle|_s$, we say that a *and* a' *are observationally indistinguishable in* M, written $a \approx_M a'$, if for all observations ϕ for sort s,

$$M[z \mapsto a] \models \phi \iff N[z \mapsto a'] \models \phi$$

Thus defined *observational indistinguishability* on M, $\approx_M \subseteq |\langle M \rangle| \times |\langle M \rangle|$, is the largest strong congruence on $\langle M \rangle$. The *observational quotient* of M, written $M/{\approx}$, is the quotient of $\langle M \rangle$ by $\approx_M$.

Theorem 5.8. Consider a CASL signature Σ. Two Σ-models are observationally equivalent if and only if their observational quotients are isomorphic.

Proof. For all CASL models M, since there is a natural strong homomorphism from $\langle M \rangle$ to $M/{\approx}$, which is a correspondence between M and $M/{\approx}$, we have that $M \equiv M/{\approx}$. Therefore, given two CASL models $M, N \in |\mathbf{Mod}(\Sigma)|$ with isomorphic observational quotients $M/{\approx}$ and $N/{\approx}$, we get $M \equiv N$.

Suppose now $M \equiv N$. Then for any closed $\Sigma^\#$-term t of a sort s, the value t_M of t in

M is defined iff the value t_N of t in N is defined. Moreover, if this is the case, then for any observation $\phi(z)$ for sort s:

$$M[z \mapsto t_M] \models \phi(z) \iff M \models \phi(t) \iff N \models \phi(t) \iff N[z \mapsto t_N] \models \phi(z)$$

It follows that for any closed $\Sigma^{\#}$-terms t and t' of a common sort s, if their values are defined in M (and hence in N as well)

$$t_M \approx_M t'_M \iff t_N \approx_N t'_N$$

Consequently, a function that for each closed $\Sigma^{\#}$-term t with defined value in M maps the equivalence class of t_M w.r.t. $\approx_M$ to the equivalence class of t_N w.r.t. $\approx_N$ is a well-defined, bijective, strong homomorphism, and hence an isomorphism, between $M/{\approx}$ and $N/{\approx}$. □

Corollary 5.9. Consider a Casl signature Σ. Σ-models M and N are observationally equivalent if and only if they have submodels with common strong quotients, that is, there exist submodels M' of M and N' of N and strong congruences $\simeq$ on M' and $\simeq'$ on N' such that the quotients of M' by $\simeq$ and of N' by $\simeq'$ are isomorphic. □

6. Observational Correctness and Stability

The observational concepts introduced in Sect. 5 above motivate a new interpretation of specifications: for any specification SP with $Sig[SP] = \Sigma$, we define its *observational interpretation* by abstracting from the standard interpretation as follows:

$$[\![SP]\!]_{\equiv} = \{M \in |\mathbf{Mod}(\Sigma)| \mid M \equiv N \text{ for some } N \in [\![SP]\!]\}.$$

Given this, the most obvious way to re-interpret correctness of local constructions (Def. 3.2) to take advantage of the observational interpretation of specifications is to modify the earlier definition by requiring $[\![SP]\!]_{\equiv} \subseteq dom(F)$ and $F([\![SP]\!]_{\equiv}) \subseteq [\![SP']\!]_{\equiv}$. This works, but misses a crucial point: when using a realization of a specification, we would like to pretend that it satisfies the specification literally, even if when actually implementing it we are permitted to supply a model that is correct only up to observational equivalence. This leads to a different notion of observational correctness of a local construction, for which we would just require $[\![SP]\!] \subseteq dom(F)$ and $F([\![SP]\!]) \subseteq [\![SP']\!]_{\equiv}$. This relaxation has a price: observationally correct local constructions do not automatically compose! The crucial insight to resolve this problem comes from (Schoett 1987), who noticed that well-behaved constructions satisfy the following *stability* property.

6.1. *Stability*

Definition 6.1 (Stability). A construction $F\colon |\mathbf{Mod}(\Sigma)| \rightharpoonup |\mathbf{Mod}(\Sigma')|$ is *stable* if it preserves observational equivalence, i.e., for any models $M, N \in |\mathbf{Mod}(\Sigma)|$ such that $M \equiv N$, if $M \in dom(F)$ then $N \in dom(F)$ and $F(M) \equiv F(N)$.

The rest of this subsection is devoted to an analysis of conditions that ensure stability of constructions when they arise via the use of local constructions, as in Sect. 3. The problem is that we want to restrict attention to conditions that are essentially local to the local constructions involved, rather than conditions that refer to all the possible global contexts in which such a construction can be used.

Let us start with the local version of the stability property for local constructions, aiming at the stability of any use of local constructions in an admissible global context.

Definition 6.2 (Local stability). A local construction F along $\iota\colon \Sigma \to \Sigma'$ is *locally stable* if for any Σ-models $M, N \in |\mathbf{Mod}(\Sigma)|$ and correspondence $\rho\colon M \bowtie N$, $M \in \mathit{dom}(F)$ if and only if $N \in \mathit{dom}(F)$ and moreover, if this is the case then there exists a correspondence $\rho'\colon F(M) \bowtie F(N)$ that extends ρ (i.e., $\rho'\big|_\iota = \rho$).

Clearly, local stability implies stability. Trivial identity constructions are locally stable, and composition of locally stable constructions is locally stable as well. Local stability is also preserved under observational equivalence of constructions:

Local constructions F_1, F_2 along $\iota\colon \Sigma \to \Sigma'$ are *observationally equivalent*, written $F_1 \equiv F_2$, if $\mathit{dom}(F_1) = \mathit{dom}(F_2)$ and for each $M \in \mathit{dom}(F_1)$ there exists a correspondence $\rho\colon F_1(M) \bowtie F_2(M)$ with reduct $\rho\big|_\iota$ being the identity on M.

Proposition 6.3. Let F_1 and F_2 be observationally equivalent local constructions along $\iota\colon \Sigma \to \Sigma'$. Then if F_1 is locally stable then so is F_2.

Proof. Consider models $M, N \in |\mathbf{Mod}(\Sigma)|$ with correspondence $\rho\colon M \bowtie N$. Suppose $M \in \mathit{dom}(F_2)$. Then $M \in \mathit{dom}(F_1)$, and so $N \in \mathit{dom}(F_1) = \mathit{dom}(F_2)$. Since F_1 is locally stable, there is a correspondence $\rho'\colon F_1(M) \bowtie F_1(N)$ with $\rho'\big|_\iota = \rho$. From $F_1 \equiv F_2$, we get correspondences $\rho_M\colon F_2(M) \bowtie F_1(M)$ and $\rho_N\colon F_1(N) \bowtie F_2(N)$ with the identity reducts $\rho_M\big|_\iota$ and $\rho_N\big|_\iota$. This yields a correspondence $(\rho_M;\rho';\rho_N)\colon F_2(M) \bowtie F_2(N)$ with reduct $(\rho_M;\rho';\rho_N)\big|_\iota = \rho$. □

Most crucially though, local stability (*unlike* stability in general) is preserved under lifting local constructions to a global application context, as usual given by the following pushout diagram:

$$\begin{tikzcd} \Sigma_G \arrow[r, "\iota'"] & \Sigma'_G \\ \Sigma \arrow[u, "\gamma"] \arrow[r, "\iota"'] & \Sigma' \arrow[u, "\gamma'"'] \end{tikzcd}$$

Lemma 6.4. If F is a locally stable construction along $\iota\colon \Sigma \to \Sigma'$ then for any signature Σ_G and admissible fitting morphism $\gamma\colon \Sigma \to \Sigma_G$, the induced global construction $F_G\colon |\mathbf{Mod}(\Sigma_G)| \rightharpoonup |\mathbf{Mod}(\Sigma'_G)|$ along $\iota'\colon \Sigma_G \to \Sigma'_G$ is locally stable as well.

Proof. Consider a correspondence $\rho_G\colon \mathcal{G} \bowtie \mathcal{H}$ between models $\mathcal{G}, \mathcal{H} \in |\mathbf{Mod}(\Sigma_G)|$. Its reduct is a correspondence $\rho_G\big|_\gamma\colon \mathcal{G}\big|_\gamma \bowtie \mathcal{H}\big|_\gamma$, so $\mathcal{G}\big|_\gamma \in \mathit{dom}(F)$ iff $\mathcal{H}\big|_\gamma \in \mathit{dom}(F)$, and consequently $\mathcal{G} \in \mathit{dom}(F_G)$ iff $\mathcal{H} \in \mathit{dom}(F_G)$. Suppose $\mathcal{G}\big|_\gamma \in \mathit{dom}(F)$. Then there exists a correspondence $\rho'\colon F(\mathcal{G}\big|_\gamma) \bowtie F(\mathcal{H}\big|_\gamma)$ with $\rho'\big|_\iota = \rho_G\big|_\gamma$. Amalgamation of ρ_G and ρ' yields a correspondence $\rho'_G\colon F_G(\mathcal{G}) \bowtie F_G(\mathcal{H})$ such that $\rho'_G\big|_{\iota'} = \rho_G$, see Lemma 5.6. □

Corollary 6.5. If F is a locally stable construction along $\iota\colon \Sigma \to \Sigma'$ then for any signature Σ_G and admissible fitting morphism $\gamma\colon \Sigma \to \Sigma_G$, the induced global construction $F_G\colon |\mathbf{Mod}(\Sigma_G)| \rightharpoonup |\mathbf{Mod}(\Sigma'_G)|$ along $\iota'\colon \Sigma_G \to \Sigma'_G$ is stable. □

This establishes a sufficient local condition (local stability) which ensures that a local construction induces a stable global construction in every possible context of use. Imposing an additional requirement on the correspondences involved yields an auxiliary notion that we will use to prove that this is both sufficient and necessary.

Given a CASL signature Σ, a correspondence $\rho\colon M \bowtie N$ is *closed* if whenever $(a,b) \in \rho$, $(a',b) \in \rho$ and $(a,b') \in \rho$, then also $(a',b') \in \rho$. The following is easy:

Proposition 6.6. For any correspondence $\rho\colon M \bowtie N$ there is a least closed correspondence $\widehat{\rho}\colon M \bowtie N$ that contains ρ. □

Consequently, two Σ-models are behaviourally equivalent iff there is a closed correspondence between them.

Theorem 6.7. For any local construction F along $\iota\colon \Sigma \to \Sigma'$, the following conditions are equivalent:

1 F is locally stable;
2 F induces a stable global construction in every possible (also infinitary) context of use, that is, for every admissible fitting morphism $\gamma\colon \Sigma \to \Sigma_G$, the induced global construction $F_G\colon |\mathbf{Mod}(\Sigma_G)| \rightharpoonup |\mathbf{Mod}(\Sigma'_G)|$ along $\iota'\colon \Sigma_G \to \Sigma'_G$ is stable; and
3 F extends closed correspondences, that is, for every closed correspondence $\widehat{\rho}\colon M \bowtie N$ in $\mathbf{Mod}(\Sigma)$, $M \in \mathit{dom}(F)$ iff $N \in \mathit{dom}(F)$, and if this is the case then there exists a closed correspondence $\widehat{\rho'}\colon F(M) \bowtie F(N)$ in $\mathbf{Mod}(\Sigma')$ that extends $\widehat{\rho}$ (i.e., $\widehat{\rho'}\big|_\iota = \widehat{\rho}$).

Proof. "$1 \Longrightarrow 2$": Cor. 6.5.

"$2 \Longrightarrow 3$": Consider a closed correspondence $\widehat{\rho}\colon M \bowtie N$ in $\mathbf{Mod}(\Sigma)$. Construct the extension Σ_G of Σ by adding:

— for each sort s in Σ and $(a,b) \in \widehat{\rho}_s$, a (total) constant $!^{a,b,s}\colon s$;
— for each sort s in Σ and $b \in |N|_s$, a predicate $?^{b,s}\colon s$; and
— for each sort s in Σ, a predicate $?^s\colon s$

and let $\gamma\colon \Sigma \to \Sigma_G$ be the signature inclusion. The admissibility of γ is easy to check. Construct now the following expansions M_G and N_G of M and N, respectively:

— for each sort s in Σ and $(a,b) \in \widehat{\rho}_s$, $!^{a,b,s}_{M_G} = a$ and $!^{a,b,s}_{N_G} = b$;
— for each sort s in Σ and $b \in |N|_s$, $?^{b,s}_{M_G}(a)$ holds iff $(a,b) \in \widehat{\rho}_s$; $?^{b,s}_{N_G}(b')$ holds iff there exists $a \in |M|_s$ such that $(a,b) \in \widehat{\rho}_s$ and $(a,b') \in \widehat{\rho}_s$;
— for each sort s in Σ and $a \in |M|_s$, $?^s_{M_G}(a)$ holds, and for each $b \in |N|_s$ $?^s_{N_G}(b)$ holds iff there exists $a \in |M|_s$ such that $(a,b) \in \widehat{\rho}_s$.

It is easy to check that $\widehat{\rho}\colon M_G \bowtie N_G$ is a correspondence: closedness of $\widehat{\rho}\colon M \bowtie N$ is needed to establish that $\widehat{\rho}$ preserves and reflects the $?^{b,s}$ predicates. Moreover, $\widehat{\rho}$ is the only correspondence between M_G and N_G: any such correspondence includes $\widehat{\rho}$ because it must preserve the $!^{a,b,s}$ constants, and it is included in $\widehat{\rho}$ because it must preserve and reflect the $?^{b,s}$ and $?^s$ predicates.

Hence, $M_G \in dom(F_G)$ iff $N_G \in dom(F_G)$, and so also $M \in dom(F)$ iff $N \in dom(F)$. Moreover, if this is the case then there is a correspondence $\rho_G: F_G(M_G) \bowtie F_G(N_G)$ in $\mathbf{Mod}(\Sigma'_G)$, and the uniqueness of the correspondence $\widehat{\rho}: M_G \bowtie N_G$ in $\mathbf{Mod}(\Sigma_G)$ implies that $\rho_G\big|_{\iota'} = \widehat{\rho}$. Consider the least closed correspondence $\widehat{\rho_G}: F_G(M_G) \bowtie F_G(N_G)$ that includes ρ_G. Then we also have $\widehat{\rho_G}\big|_{\iota'} = \widehat{\rho}$, and so we obtain $\widehat{\rho_G}\big|_{\gamma'}: F(M) \bowtie F(N)$ with $(\widehat{\rho_G}\big|_{\gamma'})\big|_{\iota} = \widehat{\rho}$.

"$3 \Longrightarrow 1$": Consider a correspondence $\rho: M \bowtie N$ in $\mathbf{Mod}(\Sigma)$. By Prop. 5.4, we have a Σ-model K and strong Σ-homomorphisms $h_M: K \to M$ and $h_N: K \to N$ such that $\rho = h_M^{-1};h_N$. Since h_M^{-1} and h_N are closed correspondences, by 3 $M \in dom(F)$ iff $K \in dom(F)$ iff $N \in dom(F)$, and if this is the case then we have correspondences $\rho_M: F(M) \bowtie F(K)$ and $\rho_N: F(K) \bowtie F(N)$ that extend h_M^{-1} and h_N respectively. Then the correspondence $\rho_M;\rho_N: F(M) \bowtie F(N)$ extends ρ. □

The following is a corollary of Lemma 5.5.

Corollary 6.8. Let $\delta: \Sigma' \to \Sigma$ be an observationally sensible derived signature morphism and $\iota: \Sigma \to \Sigma'$ be a signature morphism such that $\iota;\delta = id_\Sigma$. Then the reduct $\_\big|_\delta: |\mathbf{Mod}(\Sigma)| \to |\mathbf{Mod}(\Sigma')|$ is a local construction that is locally stable. □

The above corollary supports the point put forward in (Schoett 1987) that stable constructions are those that respect modularity in the software construction process. That is, such constructions can use the components provided by their imported parameters, but they cannot take advantage of their particular internal properties. This is the point of the requirement that δ should be observationally sensible: any branching in the code must be governed by directly observable properties. This turns (local) stability into a directive for language design, rather than a condition to be checked on a case-by-case basis: in a language with good modularization facilities, all constructions that one can code should be locally stable.

6.2. *Observational correctness*

Let us turn now again to the issue of correctness of local constructions w.r.t. given specifications.

Definition 6.9 (Observational correctness). A local construction F along $\iota: Sig[SP] \to Sig[SP']$ is *observationally correct w.r.t. SP and SP'* if for every model $M \in [\![SP]\!]$, $M \in dom(F)$ and there exists a model $M' \in [\![SP']\!]$ and correspondence $\rho': M' \bowtie F(M)$ such that $\rho'\big|_\iota$ is the identity.

We write $[\![SP \xrightarrow{\iota} SP']\!]_\equiv$ for the class of all locally stable constructions along ι that are observationally correct w.r.t. SP and SP'.

By imposing in this definition the restriction that ρ' is the identity on the carriers of the parameter sorts, we have in fact "locally" introduced a set of sorts that act as directly observable for the purposes of verification of the local construction considered.

It follows that if $F \in [\![SP \xrightarrow{\iota} SP']\!]_\equiv$ then there is some $F' \in [\![SP \xrightarrow{\iota} SP']\!]$ such that $dom(F') = dom(F)$ and for each $M \in [\![SP]\!]$, there is a correspondence $\rho: F'(M) \bowtie F(M)$

which is the identity on sorts of the form $\iota(s)$ for s in Σ. However, in general $\llbracket SP \xrightarrow{\iota} SP' \rrbracket \not\subseteq \llbracket SP \xrightarrow{\iota} SP' \rrbracket_{\equiv}$, as literally correct local constructions need not be stable. Moreover, it may happen that there are no stable observationally correct constructions, even if there are literally correct ones: that is, we may have $\llbracket SP \xrightarrow{\iota} SP' \rrbracket_{\equiv} = \emptyset$ even if $\llbracket SP \xrightarrow{\iota} SP' \rrbracket \neq \emptyset$. This was perhaps first pointed out in (Bernot 1987), in a different framework.

Counterexample 6.10. Let SP_1 have a sort s with two constants $a, b{:}\, s$, and let SP_2 enrich SP_1 by a new sort o with predicate $p : o \times o$, two (total) constants $c, d{:}\, o$ and axiom $p(c,d) \iff a = b$. Then $\llbracket SP_1 \to SP_2 \rrbracket$ is non-empty, with any construction in it mapping models satisfying $a = b$ to those that satisfy $p(c,d)$, and models satisfying $a \neq b$ to those that do not satisfy $p(c,d)$. But none of these constructions is stable!

To see this, consider any construction $F \in \llbracket SP_1 \xrightarrow{\iota} SP_2 \rrbracket$, "singleton" model $M \in \llbracket SP_1 \rrbracket$ (where $a_M = b_M$) and two-element model $N \in \llbracket SP_2 \rrbracket$ with $a_N \neq b_N$. Clearly, $M \equiv N$. However, there is no correspondence between $F(M)$ and $F(N)$: it would have to link $c_{F(M)}$ with $c_{F(N)}$ and $d_{F(M)}$ with $d_{F(N)}$, which is impossible since $F(M) \models p(c,d)$ while $F(N) \not\models p(c,d)$. $\square$

The crucial issue here is how specifications of local constructions can be used when the local constructions are lifted to an admissible global context, captured by the following pushout diagram:

$$\begin{tikzcd} \Sigma_G \arrow[r, "\iota'"] & \Sigma'_G \\ Sig[SP] \arrow[u, "\gamma"] \arrow[r, "\iota"'] & Sig[SP'] \arrow[u, "\gamma'"'] \end{tikzcd}$$

Lemma 6.11. Consider a local construction F along $\iota{:}\, Sig[SP] \to Sig[SP']$ that is observationally correct w.r.t. SP and SP', $F \in \llbracket SP \xrightarrow{\iota} SP' \rrbracket_{\equiv}$. Then, for every global signature Σ_G and admissible fitting morphism $\gamma{:}\, Sig[SP] \to \Sigma_G$, and every $\mathcal{G} \in \llbracket SP \textbf{ with } \gamma \rrbracket$ we have $\mathcal{G} \in dom(F_G)$ and there is some $\mathcal{G}' \in \llbracket SP' \textbf{ with } \gamma' \rrbracket$ such that $\mathcal{G}'|_{\iota'} = \mathcal{G}$ and $\mathcal{G}' \equiv F_G(\mathcal{G})$.

Proof. We have $\mathcal{G}|_{\gamma} \in \llbracket SP \rrbracket$, and so $\mathcal{G}|_{\gamma} \in dom(F)$ and there is $M' \in \llbracket SP' \rrbracket$ and a correspondence $\rho'{:}\, M' \bowtie F(\mathcal{G}|_{\gamma})$ with identity reduct $\rho'|_{\iota}$. Consider the Σ'_G-model $\mathcal{G}' = \mathcal{G} \oplus M'$. Then the identity $id_{\mathcal{G}}{:}\, \mathcal{G} \bowtie \mathcal{G}$ and $\rho'{:}\, M' \bowtie F(\mathcal{G}|_{\gamma})$ amalgamate to a correspondence $\rho'_G{:}\, \mathcal{G}' \bowtie F_G(\mathcal{G})$, which proves $F_G(\mathcal{G}) \equiv \mathcal{G}' \in \llbracket SP' \textbf{ with } \gamma' \rrbracket$. $\square$

If $F \in \llbracket SP \xrightarrow{\iota} SP' \rrbracket_{\equiv}$ and $\gamma{:}\, Sig[SP] \to \Sigma_G$ is admissible, then by Lemma 6.11 we obtain $\llbracket SP \textbf{ with } \gamma \rrbracket \subseteq dom(F_G)$ and $F_G(\llbracket SP \textbf{ with } \gamma \rrbracket) \subseteq \llbracket SP' \textbf{ with } \gamma' \rrbracket_{\equiv}$, and by Cor. 6.5, F_G is stable. Given two "global" specifications SP_G with $Sig[SP_G] = \Sigma_G$ and SP'_G with $Sig[SP'_G] = \Sigma'_G$, we have $F_G \in \llbracket SP_G \xrightarrow{\iota'} SP'_G \rrbracket_{\equiv}$ whenever $\llbracket SP_G \rrbracket \subseteq \llbracket SP \textbf{ with } \gamma \rrbracket_{\equiv}$ and $\llbracket SP' \textbf{ with } \gamma' \rrbracket \subseteq \llbracket SP'_G \rrbracket_{\equiv}$. But while the former requirement is quite acceptable, the latter is in fact impossible to achieve in practice since it implicitly requires that all the global requirements must follow (up to observational equivalence) from the result specification for the local construction, independent of the argument. More practical requirements are obtained by generalizing Thm. 3.4 to the observational setting:

Theorem 6.12. Given a local construction $F \in [\![SP \xrightarrow{\iota} SP']\!]_{\equiv}$, a specification SP_G with admissible fitting morphism $\gamma\colon Sig[SP] \to Sig[SP_G]$, and a specification SP'_G with $Sig[SP'_G] = \Sigma'_G$, if

(i) $[\![SP_G]\!] \subseteq [\![SP_G \textbf{ and } (SP \textbf{ with } \gamma)]\!]_{\equiv}$, and
(ii) $[\![(SP' \textbf{ with } \gamma') \textbf{ and } (SP_G \textbf{ with } \iota')]\!] \subseteq [\![SP'_G]\!]_{\equiv}$

then for every $\mathcal{G} \in [\![SP_G]\!]$, we have $\mathcal{G} \in dom(F_G)$ and $F_G(\mathcal{G}) \in [\![SP'_G]\!]_{\equiv}$, hence $F_G \in [\![SP_G \xrightarrow{\iota'} SP'_G]\!]_{\equiv}$.

Proof. Let $\mathcal{G} \in [\![SP_G]\!]$. Then $\mathcal{G} \equiv \mathcal{H}$ for some $\mathcal{H} \in [\![SP_G]\!] \cap [\![SP \textbf{ with } \gamma]\!]$ by (i). By Lemma 6.11, $F_G(\mathcal{H}) \equiv \mathcal{H}'$ for some $\mathcal{H}' \in [\![SP' \textbf{ with } \gamma']\!]$ with $\mathcal{H}'|_{\iota'} = \mathcal{H} \in [\![SP_G]\!]$. Hence $\mathcal{H}' \in [\![SP'_G]\!]_{\equiv}$ by (ii). By stability of F_G (Cor. 6.5), $\mathcal{G} \in dom(F_G)$ and $F_G(\mathcal{G}) \equiv F_G(\mathcal{H}) \equiv \mathcal{H}'$, and so $F_G(\mathcal{G}) \in [\![SP'_G]\!]_{\equiv}$. This completes the proof, since F_G is locally stable by Lemma 6.4. □

Requirement (i) is perhaps the only surprising assumption in this theorem. Note though that it straightforwardly follows from the inclusion of literal model classes $[\![SP_G]\!] \subseteq [\![SP \textbf{ with } \gamma]\!]$ (or equivalently, $[\![SP_G]\!]|_{\gamma} \subseteq [\![SP]\!]$), which is often easiest to verify. However, (i) is strictly stronger in general than the perhaps more expected $[\![SP_G]\!] \subseteq [\![SP \textbf{ with } \gamma]\!]_{\equiv}$. This weaker condition turns out to be sufficient (and is in fact equivalent to (i)) if we additionally assume that the two specifications involved are *behaviourally consistent* (Bidoit, Hennicker and Wirsing 1995), that is, closed under observational quotients. When this is not the case, then the use of this weaker condition would have to be paid for by a stronger version of (ii):

$$[\![SP' \textbf{ with } \gamma']\!]_{\equiv} \cap [\![SP_G \textbf{ with } \iota']\!] \subseteq [\![SP'_G]\!]_{\equiv},$$

which seems even less convenient to use than (i). Overall, we need a way to pass information on the global context from SP_G to SP'_G independently from the observational interpretation of the local construction and its correctness, and this must result in some inconvenience of verification on either the parameter or the result side.

7. Observational Interpretation of Architectural Specifications

In this section we discuss an observational interpretation of the architectural specifications introduced in Sect. 4. The extended static semantics remains unchanged — observational interpretation of specifications does not affect their static properties. We provide, however, a new *observational model semantics*, with judgments written as $\_ \vdash \_ \stackrel{\equiv}{\Longrightarrow} \_$.

To begin with, the effect of unit declarations has to be modified, taking into account observational interpretation of the specifications involved, as discussed in Sects. 5 and 6. The new rules follow in Fig. 4. No other modifications are necessary: all the remaining rules are the same for observational and literal model semantics. This should not be surprising: the interpretation of the constructs on unit terms remains the same, all we change is the interpretation of unit specifications. Moreover, the observational model semantics can be linked to the extended static semantics in exactly the same way as in the case of the literal model semantics: the invariants stated in Thm. 4.2 carry over

$$\frac{}{\mathcal{C} \vdash U\colon SP \stackrel{\equiv}{\Longrightarrow} \mathcal{C} \times \{U \mapsto [\![SP]\!]_{\equiv}\}}$$

$$\frac{}{\mathcal{C} \vdash U\colon SP_1 \xrightarrow{\iota} SP_2 \stackrel{\equiv}{\Longrightarrow} \mathcal{C} \times \{U \mapsto [\![SP_1 \xrightarrow{\iota} SP_2]\!]_{\equiv}\}}$$

Fig. 4. Observational model semantics — the modified rules

without change. We refrain from repeating either the unmodified rules, or Thm. 4.2 for the observational model semantics.

The fact that nearly all the rules remain the same does not mean that the two semantics quite coincide: at the point in the model semantics where verification is performed, the resulting verification conditions for literal and observational model semantics differ. Namely, in the rule for parametrized unit application, the premise

$$\text{for each } E \in \mathcal{C},\ UEv(E)\big|_{\gamma} \in dom(E(U))$$

checks whether what we can conclude about the argument ensures that it is indeed in the domain of the parametrized unit. Suppose the corresponding unit declaration was $U\colon SP_1 \xrightarrow{\iota} SP_2$. Then in the literal model semantics this requirement reduces to

$$\text{for each } E \in \mathcal{C},\ UEv(E)\big|_{\gamma} \in [\![SP_1]\!].$$

Now, in the observational model semantics, this is in fact replaced by a more permissive condition (since the parametrized units considered are locally stable, their domains are closed under observational equivalence):

$$\text{for each } E \in \mathcal{C},\ UEv(E)\big|_{\gamma} \in [\![SP_1]\!]_{\equiv}.$$

Of course, the situation is complicated by the fact that the contexts $\mathcal{C}$ from which environments are taken are different in the two semantics. In the simplest case, where the argument T is given as a unit name previously declared with a specification SP, for the literal model semantics the above verification condition amounts to $[\![SP]\!] \subseteq [\![SP_1]\!]$ while for the observational model semantics we get, as expected, $[\![SP]\!] \subseteq [\![SP_1]\!]_{\equiv}$ (which is equivalent to $[\![SP]\!]_{\equiv} \subseteq [\![SP_1]\!]_{\equiv}$).

This relaxation of verification conditions is not of merely theoretical interest: it is not difficult to find statically correct architectural specifications ASP (i.e., $\vdash ASP \rhd\!\!\!\rhd (\mathcal{C}_{st}, \Sigma)$ for some extended static context $\mathcal{C}_{st}$ and signature Σ) that are observationally correct (i.e., $\vdash ASP \stackrel{\equiv}{\Longrightarrow} (\mathcal{C}_\flat, UEv_\flat)$ for some unit context $\mathcal{C}_\flat$ and evaluator $UEv_\flat$) but *are not* literally correct (i.e., for *no* unit context $\mathcal{C}$ and evaluator UEv can we derive $\vdash ASP \Rightarrow (\mathcal{C}, UEv)$). For instance, along the lines of the discussion above, one may take

arch spec ASP =
  units $U\colon SP_1 \xrightarrow{\iota} SP_2$;
    $T : SP$
  result $U[T]$

where $Sig[SP] = Sig[SP_1]$, $[\![SP]\!] \subseteq [\![SP_1]\!]_{\equiv}$ but $[\![SP]\!] \not\subseteq [\![SP_1]\!]$.

A complete study of verification conditions for architectural specifications is beyond the scope of this paper; we refer to (Hoffman 2001; Mossakowski *et al.* 2004) for work in this direction, which still has to be combined with the observational interpretation as given by the semantics here and presented in the simpler setting of Sect. 6. In the rest of this paper we will concentrate on some aspects of the relationship between the literal and observational model semantics and on stability of the unit constructions introduced in Sect. 4.

Our first aim is to show that constructions that can be defined by architectural specifications are (locally) stable. To state this precisely, we need some more notation and terminology, as constructions are captured here by unit evaluators operating on environments rather than on individual units.

For any extended static context $\mathcal{C}_{st} = (P_{st}, \mathcal{B}_{st}, D)$, environments $E_1, E_2 \in ucx(\mathcal{C}_{st})$ are *observationally equivalent in* $\mathcal{C}_{st}$, written $E_1 \equiv_{\mathcal{C}_{st}} E_2$, if for each unit name U in $\mathcal{C}_{st}$, $E_1(U) \equiv E_2(U)$. A unit environment $E \in ucx(\mathcal{C}_{st})$ is *stable in* $\mathcal{C}_{st}$ if for each parametrized unit name U in $\mathcal{C}_{st}$, $E(U)$ is locally stable. By Prop. 6.3, the class of environments that are stable in $\mathcal{C}_{st}$ is closed under observational equivalence in $\mathcal{C}_{st}$. We write $ucx_\flat(\mathcal{C}_{st})$ for the class of all unit environments that fit $\mathcal{C}_{st}$ and are stable in $\mathcal{C}_{st}$.

A *D-coherent correspondence* between D-coherent model families $\mathcal{M}^1, \mathcal{M}^2 \in |\mathbf{Mod}(D)|$, written $\rho{:}\, \mathcal{M}^1 \bowtie \mathcal{M}^2$, is a family of correspondences $\rho_i{:}\, \mathcal{M}^1_i \bowtie \mathcal{M}^2_i$ for $i \in |D|$ such that $\rho_i = \rho_j\big|_{D(m)}$ for each $m{:}\, i \to j$ in D.

Two unit environments $E_1, E_2 \in ucx_\flat(\mathcal{C}_{st})$ are *coherently equivalent in* $\mathcal{C}_{st}$, written $E_1 \bowtie_{\mathcal{C}_{st}} E_2$, if for all parametrized unit names U in $\mathcal{C}_{st}$, $E_1(U) \equiv E_2(U)$, and there are D-coherent families of models $\mathcal{M}_1$ and $\mathcal{M}_2$ with a D-coherent correspondence $\rho{:}\, \mathcal{M}_1 \bowtie \mathcal{M}_2$ such that $\mathcal{M}_1$ and $\mathcal{M}_2$ witness E_1 and E_2 respectively in $\mathcal{C}_{st}$.

Then, given a unit context $\mathcal{C} \subseteq ucx(\mathcal{C}_{st})$, we write $Cl^{\mathcal{C}_{st}}_{\equiv}(\mathcal{C})$ for the class of all unit environments that in $\mathcal{C}_{st}$ are stable and coherently equivalent to a unit environment in $\mathcal{C}$. Clearly then $Cl^{\mathcal{C}_{st}}_{\equiv}(\mathcal{C}) \subseteq ucx_\flat(\mathcal{C}_{st})$.

Back to the stability of the constructions defined by architectural specifications: we want to show that if $\vdash ASP \rhd\!\!\rhd (\mathcal{C}_{st}, \Sigma)$ and $\vdash ASP \stackrel{\equiv}{\Longrightarrow} (\mathcal{C}_\flat, UEv_\flat)$ then the unit evaluator $UEv_\flat$ is stable, i.e., maps observationally equivalent environments to observationally equivalent models. Unfortunately, this cannot be proved by a simple induction on the structure of the unit terms involved, relying on the fact that (locally) stable constructions are closed under composition. The trouble is with amalgamation, since in general amalgamation is not stable — informally, joining the signatures of two models may introduce new observations for either or both of them, see Counterexample 5.7.

However, the key point here is that amalgamation in unit terms in architectural specifications is not used as a construction on its own, but it just identifies a new part of the global context that has been constructed earlier. Since the constructions used to build genuinely new components of the global context are locally stable, such use of amalgamation can cause no harm.

The following lemma captures the essential stability property of the unit evaluators built for unit terms by the observational model semantics.

Lemma 7.1. Assume $\mathcal{C}_{st} \vdash T \rhd\!\!\rhd (i, D')$ and $\mathcal{C}_\flat \vdash T \stackrel{\equiv}{\Longrightarrow} UEv_\flat$ with $\mathcal{C}_\flat \subseteq ucx_\flat(\mathcal{C}_{st})$, where $\mathcal{C}_{st} = (P_{st}, \mathcal{B}_{st}, D)$. The unit evaluator $UEv_\flat$ is locally stable in the following sense:

Consider any $E_1, E_2 \in \mathcal{C}_\flat$ such that $E_1 \bowtie_{\mathcal{C}_{st}} E_2$, and $\mathcal{M}_1, \mathcal{M}_2 \in |\mathbf{Mod}(D)|$ that witness E_1 and E_2 respectively in $\mathcal{C}_{st}$. Any D-coherent correspondence $\rho: \mathcal{M}^1 \bowtie \mathcal{M}^2$ can be extended to a D'-coherent correspondence $\rho': \mathcal{M}'_1 \bowtie \mathcal{M}'_2$ between model families $\mathcal{M}'_1, \mathcal{M}'_2 \in |\mathbf{Mod}(D')|$ that extend $\mathcal{M}_1$ and $\mathcal{M}_2$ respectively and satisfy $(\mathcal{M}'_1)_i = UEv_\flat(E_1)$ and $(\mathcal{M}'_2)_i = UEv_\flat(E_2)$.

Proof. By induction on the structure of the unit term. The cases when the term is a unit name or a unit reduction are trivial.

Consider the case of parametrized unit application. Using the notation as in the corresponding rules of the extended static semantics and of the (observational) model semantics in Figs. 1 and 3 respectively, consider $E_1, E_2 \in \mathcal{C}_\flat$ such that $E_1 \bowtie_{\mathcal{C}_{st}} E_2$ and a coherent correspondence $\rho: \mathcal{M}_1 \bowtie \mathcal{M}_2$ between model families $\mathcal{M}^1$, $\mathcal{M}^2$ that witness E_1 and E_2 respectively in $\mathcal{C}_{st}$. By the inductive hypothesis, ρ can be extended to a D-coherent correspondence $\rho^T: \mathcal{M}_1^T \bowtie \mathcal{M}_2^T$, where $\mathcal{M}_1^T$ extends $\mathcal{M}_1$, $\mathcal{M}_2^T$ extends $\mathcal{M}_2$, $(\mathcal{M}_1^T)_i = UEv(E_1)$ and $(\mathcal{M}_2^T)_i = UEv(E_2)$. Then, ρ^T extends to a D'-coherent correspondence $\rho': \mathcal{M}'_1 \bowtie \mathcal{M}'_2$, where $(\mathcal{M}'_1)_j = UEv(E_1)\big|_\gamma$, $(\mathcal{M}'_1)_k = E_1(U)(UEv(E_1)\big|_\gamma)$, and similarly for $\mathcal{M}'_2$ (by local stability of either $E_1(U)$ or $E_2(U)$, and the fact that $E_1(U) \equiv E_2(U)$). Now, we can extend $\mathcal{M}'_1$ and $\mathcal{M}'_2$ to D''-coherent model families $\mathcal{M}''_1$ and $\mathcal{M}''_2$, respectively, by putting $(\mathcal{M}''_1)_l = UEv(E_1)\big|_\gamma \oplus E_1(U)(UEv(E_1)\big|_\gamma)$, and similarly for $\mathcal{M}''_2$. Moreover, similarly as in Lemma 5.6, following the proof of Lemma 2.2, we can extend ρ' to a coherent correspondence $\rho'': \mathcal{M}''_1 \bowtie \mathcal{M}''_2$.

Finally, consider the case of unit amalgamation. Using the notation as in the corresponding rules of the extended static semantics and of the (observational) model semantics in Figs. 1 and 3 respectively, consider $E_1, E_2 \in \mathcal{C}_\flat$ such that $E_1 \bowtie_{\mathcal{C}_{st}} E_2$ and a coherent correspondence $\rho: \mathcal{M}_1 \bowtie \mathcal{M}_2$ between model families $\mathcal{M}_1$, $\mathcal{M}_2$ that witness E_1 and E_2 respectively in $\mathcal{C}_{st}$. By the inductive hypothesis, ρ can be extended to a D_1-coherent correspondence $\rho^{T_1}: \mathcal{M}_1^{T_1} \bowtie \mathcal{M}_2^{T_1}$, where $(\mathcal{M}_1^{T_1})$ extends $\mathcal{M}_1$, $(\mathcal{M}_2^{T_1})$ extends $\mathcal{M}_2$, $(\mathcal{M}_1^{T_1})_i = UEv_1(E_1)$ and $(\mathcal{M}_2^{T_1})_i = UEv_1(E_2)$. Similarly, ρ can be extended to a D_2-coherent correspondence $\rho^{T_2}: \mathcal{M}_1^{T_2} \bowtie \mathcal{M}_2^{T_2}$, where $(\mathcal{M}_1^{T_2})$ extends $\mathcal{M}_1$, $(\mathcal{M}_2^{T_2})$ extends $\mathcal{M}_2$, $(\mathcal{M}_1^{T_2})_i = UEv_2(E_1)$ and $(\mathcal{M}_2^{T_2})_i = UEv_2(E_2)$. Now, since D_1 and D_2 are disjoint extensions of D, ρ^{T_1} and ρ^{T_2} can be put together to form a $(D_1 \cup D_2)$-coherent correspondence between $\mathcal{M}_1^{T_1} \cup \mathcal{M}_1^{T_2}$ and $\mathcal{M}_2^{T_1} \cup \mathcal{M}_2^{T_2}$ respectively. To complete the proof we proceed as in the previous case, following Lemma 5.6 generalized as indicated in footnote 7; this is possible since the union of Casl signatures is built by taking the union of their respective sets of sort, operation and predicate names and forming the transitive closure of the union of the subsort preorders. Consequently, no new sorts, operations or predicates are added in the resulting model; everything there was constructed "earlier" while evaluating T_1 and T_2. $\square$

We can strengthen the invariant concerning the semantics of unit declarations and definitions by adding the following property:

Corollary 7.2. Let $\mathcal{C}_{st} \vdash UDD \rhd\!\!\!\rhd \mathcal{C}'_{st}$ and $\mathcal{C}_\flat \vdash UDD \overset{\equiv}{\Longrightarrow} \mathcal{C}'_\flat$ with $\mathcal{C}_\flat \subseteq ucx_\flat(\mathcal{C}_{st})$. Then

$\mathcal{C}'_\flat \subseteq ucx_\flat(\mathcal{C}'_{st})$, $\mathcal{C}'_\flat \subseteq \mathcal{C}_\flat$, and for any unit environments $E'_1, E'_2 \in \mathcal{C}'_\flat$ such that $E'_1 \equiv_{\mathcal{C}'_{st}} E'_2$, whenever $E'_1 \bowtie_{\mathcal{C}_{st}} E'_2$ then also $E'_1 \bowtie_{\mathcal{C}'_{st}} E'_2$.

Proof. This follows by easy inspection of the rules, using Lemma 7.1 for the case of unit definitions. □

Corollary 7.3. Let $\vdash UDD^+ \rhd\!\!\rhd \mathcal{C}_{st}$ and $\vdash UDD^+ \stackrel{\equiv}{\Longrightarrow} \mathcal{C}_\flat$. Then $\mathcal{C}_\flat \subseteq ucx_\flat(\mathcal{C}_{st})$ and for any unit environments $E_1, E_2 \in \mathcal{C}_\flat$ if $E_1 \equiv_{\mathcal{C}_{st}} E_2$ then also $E_1 \bowtie_{\mathcal{C}_{st}} E_2$.

Proof. For the empty extended static context $\mathcal{C}^\emptyset_{st}$, any environment in $\mathcal{C}^\emptyset$ is witnessed by the empty family of models, and so any such two environments are coherently equivalent in $\mathcal{C}^\emptyset_{st}$. Therefore, by Cor. 7.2 and an easy induction on the length of the sequence of unit declaration and definitions, for any $E_1, E_2 \in \mathcal{C}_\flat$ such that $E_1 \equiv_{\mathcal{C}_{st}} E_2$ as in the premise of the corollary, we have $E_1 \bowtie_{\mathcal{C}_{st}} E_2$. □

Corollary 7.4. If $\vdash ASP \rhd\!\!\rhd (\mathcal{C}_{st}, \Sigma)$ and $\vdash ASP \stackrel{\equiv}{\Longrightarrow} (\mathcal{C}_\flat, UEv_\flat)$ then $\mathcal{C}_\flat \subseteq ucx_\flat(\mathcal{C}_{st})$ and for any unit environments $E_1, E_2 \in \mathcal{C}_\flat$ such that $E_1 \equiv_{\mathcal{C}_{st}} E_2$, we have $UEv_\flat(E_1) \equiv UEv_\flat(E_2)$.

Proof. By Cor. 7.3 we have that for any $E_1, E_2 \in \mathcal{C}_\flat$ such that $E_1 \equiv_{\mathcal{C}_{st}} E_2$ as in the premise here, $E_1 \bowtie_{\mathcal{C}_{st}} E_2$. The conclusion follows by the stability property in Lemma 7.1. □

As already mentioned, the observational semantics is more permissive than the literal model semantics: the existence of a successful derivation of an observational meaning for an architectural specification does not in general imply that its literal model semantics is defined. Moreover, the observational semantics may "lose" some results permitted by the literal model semantics, see Counterexample 6.10. However, if an architectural specification has a literal model semantics then its observational semantics is defined as well and up to observational equivalence, nothing new is added. The following theorem captures the essential links between literal model semantics and observational model semantics.

Theorem 7.5. The following relationships between literal and observational model semantics hold:

1 If $\vdash ASP \rhd\!\!\rhd ((P_{st}, B_{st}), \Sigma)$ and $\vdash ASP \Rightarrow (\mathcal{C}, UEv)$ then $\vdash ASP \stackrel{\equiv}{\Longrightarrow} (\mathcal{C}_\flat, UEv_\flat)$ with $\mathcal{C}_\flat \subseteq Cl^{\mathcal{C}_{st}}_{\equiv}(\mathcal{C})$ and for each unit environment $E \in \mathcal{C}$ that is stable in $\mathcal{C}_{st}$, $E \in \mathcal{C}_\flat$ and $UEv_\flat(E) = UEv(E)$.

2 If $\vdash UDD^+ \rhd\!\!\rhd \mathcal{C}_{st}$ and $\vdash UDD^+ \Rightarrow \mathcal{C}$ then $\vdash UDD^+ \stackrel{\equiv}{\Longrightarrow} \mathcal{C}_\flat$, where $\mathcal{C}_\flat \subseteq Cl^{\mathcal{C}_{st}}_{\equiv}(\mathcal{C})$ and $\mathcal{C}_\flat$ contains all unit environments $E \in \mathcal{C}$ that are stable in $\mathcal{C}_{st}$.

3 If $\mathcal{C}_{st} \vdash UDD \rhd\!\!\rhd \mathcal{C}'_{st}$ and $\mathcal{C} \vdash UDD \Rightarrow \mathcal{C}'$, where $\mathcal{C} \subseteq ucx(\mathcal{C}_{st})$, then for any $\mathcal{C}_\flat \subseteq Cl^{\mathcal{C}_{st}}_{\equiv}(\mathcal{C})$ that contains all unit environments $E \in \mathcal{C}$ that are stable in $\mathcal{C}_{st}$, $\mathcal{C}_\flat \vdash UDD \stackrel{\equiv}{\Longrightarrow} \mathcal{C}'_\flat$, where $\mathcal{C}'_\flat \subseteq Cl^{\mathcal{C}'_{st}}_{\equiv}(\mathcal{C}')$ and $\mathcal{C}'_\flat$ contains all unit environments $E' \in \mathcal{C}'$ that are stable in $\mathcal{C}'_{st}$.

4 If $\mathcal{C}_{st} \vdash T \rhd\!\!\rhd (i, D')$ and $\mathcal{C} \vdash T \Rightarrow UEv$ with $\mathcal{C} \subseteq ucx(\mathcal{C}_{st})$ then for any $\mathcal{C}_\flat \subseteq Cl^{\mathcal{C}_{st}}_{\equiv}(\mathcal{C})$ that contains all unit environments $E \in \mathcal{C}$ that are stable in $\mathcal{C}_{st}$, $\mathcal{C}_\flat \vdash T \stackrel{\equiv}{\Longrightarrow} UEv_\flat$ and for $E \in \mathcal{C} \cap \mathcal{C}_\flat$, $UEv_\flat(E) = UEv(E)$.

Proof. Item 4 follows by induction on the structure of the unit term. As usual, the cases when the term is a unit name or a unit reduction are easy.

Consider the case of unit application, when the unit term is of the form $U[T \textbf{ fit } \gamma]$. Assume then $\mathcal{C} \subseteq ucx(\mathcal{C}_{st})$, $\mathcal{C}_{st} \vdash U[T \textbf{ fit } \gamma] \rhd\!\!\!\rhd (l, D'')$ and $\mathcal{C} \vdash U[T \textbf{ fit } \gamma] \Rightarrow UEv'$, with $UEv'(E) = UEv(E) \oplus E(U)(UEv(E)|_\gamma)$ for $E \in \mathcal{C}$. Consequently, all the premises of the corresponding rules (for unit application) in Figs. 1 and 3 must hold; we refer below to the notations used in the rules. Take now any $\mathcal{C}_\flat \subseteq Cl_{\equiv}^{\mathcal{C}_{st}}(\mathcal{C})$ that contains all unit environments $E \in \mathcal{C}$ that are stable in $\mathcal{C}_{st}$. By the inductive hypothesis, $\mathcal{C}_\flat \vdash T \stackrel{\equiv}{\Longrightarrow} UEv_\flat$ and for $E \in \mathcal{C} \cap \mathcal{C}_\flat$, $UEv_\flat(E) = UEv(E)$. Consider now any $E_\flat \in \mathcal{C}_\flat \subseteq Cl_{\equiv}^{\mathcal{C}_{st}}(\mathcal{C})$, with some $E \in \mathcal{C}$ such that $E_\flat \bowtie_{\mathcal{C}_{st}} E$. Then $E \in \mathcal{C} \cap \mathcal{C}_\flat$. We have $E_\flat(U) \equiv E(U)$, $UEv_\flat(E) = UEv(E)$, and since by Lemma 7.1 $UEv(E) \equiv UEv(E_\flat)$ and observational equivalence is preserved by reducts, from $UEv(E)|_\gamma \in dom(E(U))$ we obtain $UEv_\flat(E_\flat)|_\gamma \in dom(E_\flat(U))$. Thus, we can derive $\mathcal{C}_\flat \vdash U[T \textbf{ fit } \gamma] \stackrel{\equiv}{\Longrightarrow} UEv'_\flat$, where for $E_\flat \in \mathcal{C}_\flat$, $UEv'_\flat(E_\flat) = UEv_\flat(E_\flat) \oplus E_\flat(U)(UEv_\flat(E_\flat)|_\gamma)$. Now, for $E \in \mathcal{C} \cap \mathcal{C}_\flat$, since $UEv_\flat(E) = UEv(E)$, it follows that $UEv'_\flat(E) = UEv'(E)$ — which completes the proof for this case.

The proof for the case of unit amalgamation, when the unit term is of the form T_1 **and** T_2, proceeds quite similarly: assume $\mathcal{C} \subseteq ucx(\mathcal{C}_{st})$, $\mathcal{C}_{st} \vdash T_1 \textbf{ and } T_2 \rhd\!\!\!\rhd (j, D')$ and $\mathcal{C} \vdash T_1 \textbf{ and } T_2 \Rightarrow UEv$. Consequently, all the premises of the corresponding rules (for unit amalgamation) in Figs. 1 and 3 must hold; we refer below to the notations used in the rules. Take now any $\mathcal{C}_\flat \subseteq Cl_{\equiv}^{\mathcal{C}_{st}}(\mathcal{C})$ that contains all unit environments $E \in \mathcal{C}$ that are stable in $\mathcal{C}_{st}$. By the inductive hypothesis, $\mathcal{C}_\flat \vdash T_1 \stackrel{\equiv}{\Longrightarrow} UEv^1_\flat$, $\mathcal{C}_\flat \vdash T_2 \stackrel{\equiv}{\Longrightarrow} UEv^2_\flat$, and for $E \in \mathcal{C} \cap \mathcal{C}_\flat$, $UEv^1_\flat(E) = UEv_1(E)$ and $UEv^2_\flat(E) = UEv_2(E)$. Then $\mathcal{C}_\flat \vdash T_1 \textbf{ and } T_2 \stackrel{\equiv}{\Longrightarrow} UEv_\flat$, where for $E_\flat \in \mathcal{C}_\flat$, $UEv_\flat(E_\flat)$ amalgamates $UEv^1_\flat(E_\flat)$ and $UEv^2_\flat(E_\flat)$. Clearly now, by the definition of UEv in the model semantics, for $E \in \mathcal{C} \cap \mathcal{C}_\flat$, since $UEv^1_\flat(E) = UEv_1(E)$ and $UEv^2_\flat(E) = UEv_2(E)$, we conclude that $UEv_\flat(E) = UEv(E)$ — which completes the proof of item 4.

Item 3 follows by inspection of the rules; the cases of unit declarations are easy. The case of unit definition relies on item 4 as follows. Assume that $\mathcal{C} \subseteq ucx(\mathcal{C}_{st})$ and $\mathcal{C}$ is closed in $\mathcal{C}_{st} = (P_{st}, \mathcal{B}_{st}, D)$. To derive $\mathcal{C}_{st} \vdash UDD \rhd\!\!\!\rhd \mathcal{C}'_{st}$ and $\mathcal{C} \vdash UDD \Rightarrow \mathcal{C}'$, where UDD is of the form $U = T$, we must have $(P_{st}, \mathcal{B}_{st}, D) \vdash T \rhd\!\!\!\rhd (i, D')$, $U \notin (dom(P_{st}) \cup dom(\mathcal{B}_{st}))$, and $\mathcal{C} \vdash T \Rightarrow UEv$, with $\mathcal{C}'_{st} = (P_{st}, \mathcal{B}_{st} + \{U \mapsto i\}, D')$ and $\mathcal{C}' = \{E + \{U \mapsto UEv(E)\} \mid E \in \mathcal{C}\}$. Take now any $\mathcal{C}_\flat \subseteq Cl_{\equiv}^{\mathcal{C}_{st}}(\mathcal{C})$ that contains all unit environments $E \in \mathcal{C}$ that are stable in $\mathcal{C}_{st}$. By item 4, $\mathcal{C}_\flat \vdash T \stackrel{\equiv}{\Longrightarrow} UEv_\flat$ and for $E \in \mathcal{C} \cap \mathcal{C}_\flat$, $UEv_\flat(E) = UEv(E)$. Hence, $\mathcal{C}_\flat \vdash U = T \Rightarrow \mathcal{C}'_\flat$ with $\mathcal{C}'_\flat = \{E_\flat + \{U \mapsto UEv_\flat(E_\flat)\} \mid E_\flat \in \mathcal{C}_\flat\}$. To see that $\mathcal{C}'_\flat \subseteq Cl_{\equiv}^{\mathcal{C}'_{st}}(\mathcal{C}')$, consider any $E_\flat \in \mathcal{C}_\flat \subseteq Cl_{\equiv}^{\mathcal{C}_{st}}(\mathcal{C})$, with some $E \in \mathcal{C}$ such that $E_\flat \bowtie_{\mathcal{C}_{st}} E$. By Lemma 7.1, $E_\flat + \{U \mapsto UEv_\flat(E_\flat)\}$ is coherently equivalent in $\mathcal{C}'_{st}$ to $E + \{U \mapsto UEv_\flat(E)\}$, which is the same as $E + \{U \mapsto UEv(E)\}$. This shows that $E_\flat + \{U \mapsto UEv_\flat(E_\flat)\}$ is indeed in $Cl_{\equiv}^{\mathcal{C}'_{st}}(\mathcal{C}')$. Finally, if for some $E \in \mathcal{C}$, $E + \{U \mapsto UEv(E)\}$ is stable in $\mathcal{C}'_{st}$, then E is stable in $\mathcal{C}_{st}$ and so is in $\mathcal{C}_\flat$. Since then $UEv_\flat(E) = UEv(E)$ by item 4, we also have that $E + \{U \mapsto UEv(E)\}$ is in $\mathcal{C}'_\flat$.

Item 2 follows from item 3 by an easy induction on the length of the sequence of unit declarations and definitions. To start, notice that every environment in $\mathcal{C}^\emptyset$ is stable in the empty static context $\mathcal{C}^{\emptyset}_{st}$ and is witnessed in $\mathcal{C}^{\emptyset}_{st}$ by the empty family of models; hence, $\mathcal{C}^{\emptyset} = Cl^{\mathcal{C}^{\emptyset}_{st}}_{\equiv}(\mathcal{C}^{\emptyset})$.

Item 1 now follows easily: to derive the assumptions for ASP of the form **arch spec** UDD^+ **result** T, we must have $\vdash UDD^+ \rhd\!\!\!\rhd \mathcal{C}_{st}$ and $\vdash UDD^+ \Rightarrow \mathcal{C}$, as well as $\mathcal{C}_{st} \vdash T \rhd\!\!\!\rhd$ and $\mathcal{C} \vdash T \Rightarrow UEv$, with $(P_{st}, B_{st}) = ctx(\mathcal{C}_{st})$ and $\Sigma = D(i)$. By item 2 we thus have $\vdash UDD^+ \stackrel{\equiv}{\Longrightarrow} \mathcal{C}_\flat$, where $\mathcal{C}_\flat \subseteq Cl^{\mathcal{C}_{st}}_{\equiv}(\mathcal{C})$ and $\mathcal{C}_\flat$ contains all unit environments $E \in \mathcal{C}$ that are stable in $\mathcal{C}_{st}$. By item 4 in turn, $\mathcal{C}_\flat \vdash T \stackrel{\equiv}{\Longrightarrow} UEv_\flat$ and for each unit environment $E \in \mathcal{C}$ stable in $\mathcal{C}_{st}$, $UEv_\flat(E) = UEv(E)$ (since $E \in \mathcal{C} \cap \mathcal{C}_\flat$ then). $\square$

Corollary 7.6. If $\vdash ASP \rhd\!\!\!\rhd (\mathcal{C}_{st}, \Sigma)$ and $\vdash ASP \Rightarrow (\mathcal{C}, UEv)$ then $\vdash ASP \stackrel{\equiv}{\Longrightarrow} (\mathcal{C}_\flat, UEv_\flat)$, where for every $E_\flat \in \mathcal{C}_\flat$ there exists $E \in \mathcal{C}$ such that $E_\flat \equiv_{\mathcal{C}_{st}} E$ and $UEv_\flat(E_\flat) \equiv UEv(E)$.

Proof. Given the assumptions, by Thm. 7.5, $\vdash ASP \stackrel{\equiv}{\Longrightarrow} (\mathcal{C}_\flat, UEv_\flat)$ with $\mathcal{C}_\flat \subseteq Cl^{\mathcal{C}_{st}}_{\equiv}(\mathcal{C})$ and for each $E \in \mathcal{C}$ that is stable in $\mathcal{C}_{st}$, $E \in \mathcal{C}_\flat$ and $UEv_\flat(E) = UEv(E)$. Hence, for each $E_\flat \in \mathcal{C}_\flat$ there is a stable environment $E \in \mathcal{C}$ such that $E_\flat \bowtie_{\mathcal{C}_{st}} E$ and $UEv(E) = UEv_\flat(E)$. It follows that $E_\flat \equiv_{\mathcal{C}_{st}} E$ and, by Cor. 7.4, $UEv_\flat(E) \equiv UEv_\flat(E_\flat)$, which yields $UEv(E) \equiv UEv_\flat(E_\flat)$. $\square$

8. Example

The following example illustrates some of the points in the paper. The notation of CASL is hopefully understandable without further explanation; otherwise see (CoFI 2004).

We start with a simple specification of sets of strings; we will not go into any details of a specification of strings, just remarking that any standard specification would typically be monomorphic (with a unique model, up to isomorphism) and would certainly provide the equality predicate for strings.

spec STRING = **sort** *String*
 ...
 pred $eq_S : String \times String$;
 axiom $\forall s, s' : String \bullet eq_S(s, s') \iff s = s'$
 ...

spec STRINGSET = STRING
 then sort *Set*
 ops $empty : Set$;
 $add : String \times Set \to Set$
 pred $present : String \times Set$
 $\forall\ s, s' : String, t : Set$
 • $add(s, add(s, t)) = add(s, t)$
 • $add(s, add(s', t)) = add(s', add(s, t))$
 • $\neg present(s, empty)$
 • $present(s, add(s, t))$
 • $s \neq s' \Longrightarrow (present(s, add(s', t)) \iff present(s, t))$

We now provide a more elaborate version of the requirements this specification captures, introducing the idea of using a hash table implementation of sets.

spec INT = **sort** *Int*
...
pred $eq_N : Int \times Int$;
axiom $\forall n, n' : Int \bullet eq_N(n, n') \iff n = n'$
...

spec ELEM = **sort** *Elem*

spec ARRAY[ELEM] = ELEM **and** INT
then sort $Array[Elem]$
ops $empty : Array[Elem]$;
$put : Int \times Elem \times Array[Elem] \to Array[Elem]$;
$take : Int \times Array[Elem] \to ?\ Elem$
pred $used : Int \times Array[Elem]$
$\forall\ i, j : Int; e, e' : Elem; a : Array[Elem]$
- $i \neq j \Longrightarrow put(i, e', put(j, e, a)) = put(j, e, put(i, e', a))$
- $put(i, e', put(i, e, a)) = put(i, e', a)$
- $\neg used(i, empty)$
- $used(i, put(i, e, a))$
- $i \neq j \Longrightarrow (\ used(i, put(j, e, a)) \iff used(i, a)\ )$
- $take(i, put(i, e, a)) = e$

spec ELEMKEY = ELEM **and** INT
then op $hash : Elem \to Int$

spec HASHTABLE[ELEMKEY] = ELEMKEY **and** ARRAY[ELEM]
then ops $add : Elem \times Array[Elem] \to Array[Elem]$;
$putnear : Int \times Elem \times Array[Elem] \to Array[Elem]$
preds $present : Elem \times Array[Elem]$
$isnear : Int \times Elem \times Array[Elem]$
$\forall\ i : Int; e : Elem; a : Array[Elem]$
- $add(e, a) = putnear(hash(e), e, a)$
- $\neg used(i, a) \Longrightarrow putnear(i, e, a) = put(i, e, a)$
- $used(i, a) \wedge take(i, a) = e \Longrightarrow putnear(i, e, a) = a$
- $used(i, a) \wedge take(i, a) \neq e \Longrightarrow$
  $putnear(i, e, a) = putnear(succ(i), e, a)$
- $present(e, a) \iff isnear(hash(e), e, a)$
- $\neg used(i, a) \Longrightarrow \neg isnear(i, e, a)$
- $used(i, a) \wedge take(i, a) = e \Longrightarrow isnear(i, e, a)$
- $used(i, a) \wedge take(i, a) \neq e \Longrightarrow$
  $(isnear(i, e, a) \iff isnear(succ(i), e, a))$

spec STRINGKEY = STRING **and** INT
then op $hash : String \to Int$

spec STRINGHASHTABLE =
HASHTABLE[STRINGKEY] **with** $Array[String] \mapsto Set$
reveal $String, Set, empty, add, present$

STRINGHASHTABLE does not literally ensure all the requirements imposed by the original specification STRINGSET: the second axiom (commutativity of adding elements to a set) fails in some models of STRINGHASHTABLE. Still, it is easy to check that

$[\![\text{STRINGHASHTABLE}]\!] \subseteq [\![\text{STRINGSET}]\!]_{\equiv}$ and so every future (observationally-correct) realisation of STRINGHASHTABLE is an observationally-correct realisation of STRINGSET.[10]

STRINGHASHTABLE is structured in a fairly natural way, building on a generic specification of arrays that is presumably already available, and including a generic specification of hash tables that may be reused in the future.

However, the structure of STRINGHASHTABLE must not be viewed as an obligatory prescription of the structure of the final implementation. For example, we may decide to adopt the different structure given below by the architectural specification STRINGHASHTABLEDESIGN.

The architectural specification uses the CASL construct **given** to mark units that are imported by other units. Formally, a sequence of declarations like

N : INT; S : STRING;
SK : STRINGKEY **given** S, N;

abbreviates

N : INT; S : STRING;
SK' : INT $\times$ STRING $\rightarrow$ STRINGKEY;
$SK = SK'[N][S]$;

where a new generic construction SK' is introduced and immediately applied to the imported units.

arch spec STRINGHASHTABLEDESIGN =
  units N : INT;
      S : STRING;
      SK : STRINGKEY **given** S, N;
      A : ELEM $\rightarrow$ ARRAY[ELEM] **given** N;
      $ASK = A[SK$ **fit** $Elem \mapsto String]$;
      HT : STRINGHASHTABLE
          given {ASK **with** $Array[String] \mapsto Set$}
  result HT **reveal** $String, Set, empty, add, present$

The above architectural specification captures a modular design of the system to be built as follows. Components N and S are to be defined, implementing specifications INT and STRING, respectively. Presumably, these would be predefined in any practical programming language. Then, N and S are put together and extended by a definition of a hash function *hash*, yielding a new component SK. However, as explained above, the **given** notation used here really means that we are to provide a construction (a generic unit SK') that yields such a component for any realisations of INT and STRING. Another component to be provided is a generic unit A to implement arrays indexed by integers and storing data of any sort (*Elem*, to be instantiated when A is applied to an argument component). Again, this is to be given by a construction A' that works for any implementation of INT, but then is instantiated with the specific implementation given by N. This is then used to build a component ASK, that implements arrays of strings (with a hash function) by instantiating A with SK. In turn, ASK (with the

[10] Note that dropping the first two axioms in STRINGSET yields a specification with a class of models that coincides with $[\![\text{STRINGSET}]\!]_{\equiv}$ — in fact, we could have started with such an observationally-closed version of the specification, making no use of observational correctness at this stage yet.

main sort renamed to *Set* to fit the top level names given in the original requirement specification) will be extended to a component implementing StringHashTable — again, this is to be built via a construction HT', independently of the details of ASK, for an arbitrary implementation of Array[StringKey]. Finally, the overall result will be given by exporting from this component only the required sorts, operations and predicate.

Notice that the structure here differs in an essential way from the structure of StringHashTable, since we have chosen to forego genericity of hash tables (for arbitrary elements), implementing them for the special case of strings.

Further development might lead to a final implementation in Standard ML, including the following modules. The task of extracting Standard ML signatures (`ARRAY_SIG` etc., using boolean functions for predicates) from the corresponding Casl signatures of the specifications given above is left for the reader. We assume though that the implementations `N` of Int and `S` of String, which we do not spell out here, use the Standard ML built-in types `int` and `string`, respectively. These are so-called *equality types* in Standard ML, and come with the built-in (infix) equality function `_=_` which should replace eq_N and eq_S in the corresponding Standard ML signatures. We also omit a component `SK` that implements a hash function `hash`; any total function from strings to integers will do, although of course a good hash function will produce an even distribution of hash values. We compress consecutive instantiations of `A'` (first to `N` and then to `SK`) into a single functor application. Finally, we will incorporate the final adjustment to the overall result signature (the **reveal** construct in the result unit in StringHashTableDesign) and the renaming of arrays to sets (in the **given** part of `HT`) directly into the definition of the functor `HT'` used to build the resulting hash table of strings.

```
functor A'(structure N: INT_SIG and E : ELEM_SIG) : ARRAY_SIG =
  struct
    open N E
    type array = int -> elem
    exception unused
    fun empty(i) = raise unused
    fun put(i,e,a)(j) = if i=j then e else a(j)
    fun take(i,a) = a(i)
    fun used(i,a) = (a(i); true) handle unused => false
  end

structure ASK =
  struct
    structure Astring =
       A'(structure N=N and E=struct type elem=SK.string end)
    open Astring
    open SK
  end

functor HT'(structure ASK: ASK_SIG) : STRING_HASH_TABLE_SIG =
```

```
struct
  open ASK
  type set = array
  fun putnear(i,s,t) =
     if used(i,t)
     then if take(i,t)=s then t else putnear(i+1,s,t)
     else put(i,s,t)
  fun add(s,t) = putnear(hash(s),s,t)
  fun isnear(i,s,t) =
     used(i,t) andalso (take(i,t)=s orelse isnear(i+1,s,t))
  fun present(s,t) = isnear(hash(s),s,t)
end

structure HT = HT'(structure ASK=ASK)
```

The functor `A'` is literally correct with respect to INT **and** ELEM and ARRAY[ELEM]. To be more precise, the semantic function on the models determined by `A'` extends any model in $[\![$INT **and** ELEM$]\!]$ to a model in $[\![$ARRAY[ELEM]$]\!]$ such that $[\![$`A'`$]\!] \in [\![$INT **and** ELEM$\xrightarrow{\iota}$ARRAY[ELEM]$]\!]$, where ι is the obvious signature inclusion. Similarly, the structure `HT` satisfies the axioms of STRINGHASHTABLE literally (at least on the reachable part, and assuming the use of extensional equality on functions).

The reader might want to check that STRINGHASHTABLEDESIGN is a statically correct architectural specification:[11] we can derive

$$\vdash \text{STRINGHASHTABLEDESIGN} \rhd\!\!\!\rhd ((P_{st}, B_{st}), \Sigma)$$

where P_{st} binds the generic units declared in STRINGHASHTABLE (including those implicitly introduced by expanding the **given** construct for imports), B_{st} maps the non-generic unit names in STRINGHASHTABLE to their signatures, and Σ is the signature of the result unit (the signature of STRINGHASHTABLE). Moreover, the (literal) model semantics works as well, so that we have

$$\vdash \text{STRINGHASHTABLEDESIGN} \Rightarrow (\mathcal{C}, \mathit{UEv}).$$

Here, the context $\mathcal{C}$ contains all environments that map unit names declared and defined in STRINGHASHTABLEDESIGN to their realisations so that declared units satisfy their specifications and the defined units are built from the units given in the environment as prescribed by their respective definitions. Then, the unit evaluator *UEv* maps any such environment in $\mathcal{C}$ to a model as determined by the result unit definition. In particular, the environment determined by the Standard ML functor and structure definitions given above is in $\mathcal{C}$, and *UEv* maps it to the expected system realisation.

However, even though the above functor `A'` implementing arrays is correct, we might want to use quite a different array implementation, for instance because it is given as a highly optimised module in a library. Various useful "tricks" in the code then might

[11] Using, for instance, the HETS tool, see `www.informatik.uni-bremen.de/cofi/hets/`.

be expected. Here is an example where each entry in the array includes its history of updates:

```
functor Atrick(structure N: INT_SIG and E : ELEM_SIG) : ARRAY_SIG =
  struct
    open E
    type array = int -> elem list
    fun empty(i) = nil
    fun put(i,e,a)(j) = if i=j then e::a(j) else a(j)
    fun take(i,a) = let val e::_=a(i) in e end
    fun used(i,a) = not(null a(i))
  end
```

Then, `Atrick` given here is not literally correct with respect to INT **and** ELEM and ARRAY[ELEM], since it violates the axiom $put(i, e', put(i, e, a)) = put(i, e', a)$, but it is observationally correct: $[\![\texttt{Atrick}]\!] \in [\![\textsc{Int}\ \textbf{and}\ \textsc{Elem} \xrightarrow{\iota} \textsc{Array}[\textsc{Elem}]]\!]_{\equiv}$. Similarly, the extra flexibility that observational correctness offers would allow us for instance to change the code for `HT'` to count the number of insertions of each string, yielding a new functor `HTtrick'`. The structure

```
structure HTtrick = HTtrick'(structure ASK=ASK)
```

violates the axiom $used(i, a) \wedge take(i, a) = s \implies putnear(i, s, a) = a$, but again it is observationally correct: $[\![\texttt{HTtrick}]\!] \in [\![\textsc{StringHashTable}]\!]_{\equiv}$.

The unit environment determined by `Atrick'` and `HTtrick'` is not in the context $\mathcal{C}$ given by the literal model semantics of STRINGHASHTABLE. However, under the observational semantics, we have:

$$\vdash \textsc{StringHashTableDesign} \overset{\equiv}{\Longrightarrow} (\mathcal{C}_\flat, \mathit{UEv}_\flat),$$

where $\mathcal{C}_\flat$ contains the environment that is determined by `Atrick'` and `HTtrick'`. Moreover, $\mathit{UEv}_\flat$ (which essentially coincides with UEv given by the literal model semantics above, but works on a different domain) maps such an environment to a model of the whole system that is an observationally correct realisation of the original specification STRINGHASHTABLE, as expected.

The Standard ML functors above define locally stable constructions: they respect encapsulation since they do not use any properties of their arguments other than what is spelled out in their parameter signatures. Indeed, all closed functors (which do not refer to external structure definitions) in Standard ML define locally stable constructions.

Let us now go back to the idea inherent in the structure of the specification STRINGHASHTABLE, and try to build our implementation using a generic construction for hash tables. That structure may be captured by the following architectural specification:

arch spec STRINGHASHTABLEDESIGN′ =
  units N : INT;
    A : ELEM → ARRAY[ELEM] **given** N;
    HT_{gen} : ELEMKEY × ARRAY[ELEM] → HASHTABLE[ELEMKEY];
    S : STRING;
    SK : STRINGKEY **given** S, N;
  result $HT_{gen}[SK$ **fit** $Elem \mapsto String][A[S]]$ **with** $Array[String] \mapsto Set$
    reveal $String, Set, empty, add, present$

This is a correct architectural specification again, and indeed we get:

$$\vdash \text{STRINGHASHTABLEDESIGN}' \rhd\!\!\!> ((P'_{st}, B'_{st}), \Sigma)$$
$$\vdash \text{STRINGHASHTABLEDESIGN}' \Rightarrow (\mathcal{C}', \mathit{UEv}')$$
$$\vdash \text{STRINGHASHTABLEDESIGN}' \stackrel{\equiv}{\Longrightarrow} (\mathcal{C}'_\flat, \mathit{UEv}'_\flat)$$

The extended static semantics and the literal model semantics work as expected (we encourage the reader to try to describe the resulting contexts). However, perhaps unexpectedly, we get $\mathcal{C}'_\flat = \emptyset$ — the above architectural specification is observationally inconsistent! The trouble is, of course, with the specification of generic hash tables. One might try to implement it as follows:

```
functor HTgen
    (structure EK : ELEM_KEY_SIG and A : ARRAY_ELEM_KEY_SIG
         sharing type EK.elem=A.elem) : HASH_TABLE_ELEM_KEY_SIG =
  struct
    open EK A
    fun putnear(i,e,a) =
       if used(i,a)
       then if take(i,a)=e then a else putnear(i+1,e,a)
       else put(i,e,a)
    fun add(e,a) = putnear(hash(e),e,a)
    fun isnear(i,e,a) =
       used(i,a) andalso (take(i,a)=e orelse isnear(i+1,e,a))
    fun present(e,a) = isnear(hash(e),e,a)
  end
```

Unfortunately, the construction defined by `HTgen` is not locally stable, and in fact `HTgen` is not correct code in Standard ML, since it requires equality on `elem` (in `take(i,a)=e`) which is not provided by `ELEM_KEY_SIG`. This problem is not accidental: there is no locally stable construction, and hence no Standard ML functor, satisfying the required specification. Consequently, there are no stable environments in context $\mathcal{C}'$ resulting from the literal model semantics — hence the observational inconsistency of STRINGHASHTABLEDESIGN′ ($\mathcal{C}'_\flat = \emptyset$). Even though what is a reasonable structure for the requirements specification, as expressed in STRINGHASHTABLE, led to an inappropriate modular design STRINGHASHTABLEDESIGN′, this is in fact good news. While allowing for a more relaxed interpretation of the axioms in (result) specifications as long as their observable consequences are ensured, the observational semantics marked as inconsistent a specification that cannot be implemented in a reasonable programming language in which no tricky means are available to violate the modular structure.

Of course, this does not mean that there is no good design that would require a generic implementation of hash tables. A simple way to achieve this would be to modify the above architectural specification to add "equality" on *Elem* by introducing an equality predicate (for instance, in ElemKey). Notice that this is different from requiring *Elem* to be an equality type as in Standard ML, since this predicate is not necessarily constrained to be interpreted as the identity. Consequently, we should then use this predicate, rather than identity, to compare elements stored in HashTable. One point of architectural specifications is that such change of structure is an important design decision that deserves to be recorded explicitly. The new specifications would be as follows:

spec ElemKeyEq = Elem **and** Int
  then op $hash : Elem \to Int$;
    pred $eq_E : Elem \times Elem$

spec HashTableEq[ElemKeyEq] = ElemKeyEq **and** Array[Elem]
  then ops $add : Elem \times Array[Elem] \to Array[Elem]$;
    $putnear : Int \times Elem \times Array[Elem] \to Array[Elem]$
  preds $present : Elem \times Array[Elem]$
    $isnear : Int \times Elem \times Array[Elem]$
  $\forall\ i : Int; e : Elem; a : Array[Elem]$
  - $add(e, a) = putnear(hash(e), e, a)$
  - $\neg used(i, a) \Longrightarrow putnear(i, e, a) = put(i, e, a)$
  - $used(i, a) \wedge eq_E(take(i, a), e) \Longrightarrow putnear(i, e, a) = a$
  - $used(i, a) \wedge \neg eq_E(take(i, a), e) \Longrightarrow$
    $putnear(i, e, a) = putnear(succ(i), e, a)$
  - $present(e, a) \iff isnear(hash(e), e, a)$
  - $\neg used(i, a) \Longrightarrow \neg isnear(i, e, a)$
  - $used(i, a) \wedge eq_E(take(i, a), e) \Longrightarrow isnear(i, e, a)$
  - $used(i, a) \wedge \neg eq_E(take(i, a), e) \Longrightarrow$
    $(isnear(i, e, a) \iff isnear(succ(i), e, a))$

The architectural design then might look as follows:

arch spec StringHashTableDesignEq =
  units N : Int;
    A : Elem → Array[Elem] **given** N;
    HT_{gen} : ElemKeyEq × Array[Elem] → HashTableEq[ElemKeyEq];
    S : String;
    SK : StringKey **given** S, N;
  result $HT_{gen}[SK$ **fit** $Elem \mapsto String][A[S]]$ **with** $Array[String] \mapsto Set$
    reveal $String, Set, empty, add, present$

The following Standard ML functor would then provide a generic implementation of hash tables for any type of elements with an equality function, yielding a locally stable construction that is (observationally) correct with respect to ElemKeyEq **and** Array[Elem] and HashTableEq[ElemKeyEq]:

```
functor HTEQgen
    (structure EK : ELEM_KEY_EQ_SIG and A : ARRAY_ELEM_KEY_SIG
```

```
       sharing type EK.elem=A.elem) : HASH_TABLE_ELEM_KEY_EQ_SIG =
struct
  open EK A
  fun putnear(i,e,a) =
     if used(i,a)
     then if eq_E(take(i,a),e) then a else putnear(i+1,e,a)
     else put(i,e,a)
  fun add(e,a) = putnear(hash(e),e,a)
  fun isnear(i,e,a) =
     used(i,a) andalso (eq_E(take(i,a),e) orelse isnear(i+1,e,a))
  fun present(e,a) = isnear(hash(e),e,a)
end
```

9. Conclusions and Further Work

The overall goal of this paper is to provide an observational view of CASL specifications that supports observational refinement of specifications in combination with CASL-style architectural design. This is achieved, and spelled out in detail for a simplified version of CASL architectural specifications. Extending this to full CASL architectural specifications (by allowing multiple parameters for parametrized units, adding unit translations, etc.) is straightforward. Imports of units defined by arbitrary unit expressions are the only potential source of difficulty. But the methodologically well-justified case of this, where the import can be given an explicit specification, is easily dealt with as in Sect. 8.

Although we have worked in the specific setting of CASL signatures and models, formulated as an institution in Sect. 2, it should be clear that much of the above applies to a wide range of institutions. Rather than attempting to spell out the appropriate notion of "institution with extra structure", let us just remark that surprisingly little appears to be required. A notion of *observational model morphisms* that is closed under composition and reduct, plus some extra categorical structure to identify "correspondences" as certain spans of such morphisms, seems necessary and sufficient to formulate most of the material presented. The need for additional structure is obviated by the fact that the technical development makes no reference to a set of observable sorts, in contrast to standard approaches to observational interpretation of specifications. In the context of CASL (where one can treat a sort as observable by introducing an "equality predicate" on it) this is adequate. It may well not be adequate in institutions of much more limited expressive power, but it is not clear that such institutions are of genuine practical importance. Links with indistinguishability relations via factorization properties, like Thm. 5.8, may require the richer context of *concrete institutions*, where model categories are equipped with concretization structure subject to a number of technical requirements as in (Bidoit and Tarlecki 1996), or alternatively may follow the ideas of (Popescu and Roşu 2005).

A challenging issue is now to understand how far the concepts developed for our somewhat simplified view of software components as local constructions on CASL models can be inspiring for a more general view of components involving some form of external communication. While this is clearly beyond the scope of this paper, we nevertheless imagine that a promising direction of future research would be to look for an adequate counterpart of (local) stability in this more general setting.

Acknowledgements. Our thanks to the CoFI Language Design and Semantics Task Groups for many discussions and opportunities to present and improve our ideas on architectural specifications, and thanks in particular to Piotr Hoffman, Bartek Klin, Till Mossakowski and Lutz Schröder for collaboration on their semantics. Thanks also to one of the anonymous reviewers for careful checking and perceptive comments. This work has been partially supported by European projects IST-2001-32747 AGILE (AT), IST-2005-015905 MOBIUS (DS, AT) and IST-2005-016004 SENSORIA (AT), the EPSRC-funded ReQueST project (DS), the British–Polish Research Partnership Programme (DS, AT), and Visiting Professorships at ENS de Cachan (AT).

References

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).

E. Astesiano, B. Krieg-Brückner and H.-J. Kreowski, eds. *Algebraic Foundations of Systems Specification*. Springer (1999).

H. Baumeister, M. Cerioli, A. Haxthausen, T. Mossakowski, P.D. Mosses, D. Sannella and A. Tarlecki. CASL Semantics. In: (Mosses 2004), 115–273.

G. Bernot. Good functors ... are those preserving philosophy! *Proc. 2nd Summer Conf. on Category Theory and Computer Science CTCS'87*, Springer LNCS 283, 182–195 (1987).

M. Bidoit and R. Hennicker. A general framework for modular implementations of modular systems. *Proc. 4th Int. Conf. on Theory and Practice of Software Development TAPSOFT'93*, Springer LNCS 668, 199–214 (1993).

M. Bidoit and R. Hennicker. Modular correctness proofs of behavioural implementations. *Acta Informatica* 35(11):951–1005 (1998).

M. Bidoit and R. Hennicker. Proving behavioral refinements of COL-specifications. *Algebra, Meaning and Computation: Essays Dedicated to Joseph A. Goguen on the Occasion of his 65th Birthday*. Springer LNCS 4060, 333–354 (2006).

M. Bidoit, R. Hennicker and M. Wirsing. Behavioural and abstractor specifications. *Science of Computer Programming* 25:149–186 (1995).

M. Bidoit and P.D. Mosses. CASL *User Manual*. Springer LNCS 2900 (IFIP Series), 2004.

M. Bidoit, D. Sannella and A. Tarlecki. Architectural specifications in CASL. *Formal Aspects of Computing*, 13:252–273 (2002a).

M. Bidoit, D. Sannella and A. Tarlecki. Global development via local observational construction steps. *Proc. 27th Intl. Symp. on Mathematical Foundations of Computer Science, MFCS'02*. Springer LNCS 2420, 1–24 (2002b).

M. Bidoit, D. Sannella and A. Tarlecki. Toward component-oriented formal software development: an algebraic approach. *Proc. 9th Monterey Workshop, Radical Innovations of Software and Systems Engineering in the Future*, Venice, October 2002. Springer LNCS 2941, 75–90 (2004).

M. Bidoit and A. Tarlecki. Behavioural satisfaction and equivalence in concrete model categories. *Proc. 20th Coll. on Trees in Algebra and Computing CAAP'96*, Linköping, Springer LNCS 1059, 241–256 (1996).

P. Burmeister. *A Model Theoretic Approach to Partial Algebras*. Akademie Verlag, Berlin (1986).

R. Burstall and J. Goguen. The semantics of Clear, a specification language. *Proc. Advanced Course on Abstract Software Specifications*, Copenhagen. Springer LNCS 86, 292–332 (1980).

The CoFI Language Design Group. CASL Summary. B. Krieg-Brückner and P.D. Mosses, eds. In: (Mosses 2004), 3–74.

H. Ehrig and H.-J. Kreowski. Refinement and implementation. In: (Astesiano, Krieg-Brückner and Kreowski 1999), 201–242.

H. Ehrig, H.-J. Kreowski, B. Mahr and P. Padawitz. Algebraic implementation of abstract data types. *Theoretical Computer Science* 20:209–263 (1982).

H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I: Equations and Initial Semantics*. Springer (1985).

H. Ganzinger. Parameterized specifications: parameter passing and implementation with respect to observability. *ACM Transactions on Programming Languages and Systems* 5:318–354 (1983).

V. Giarratana, F. Gimona and U. Montanari. Observability concepts in abstract data type specifications. *Proc. 5th Intl. Symp. on Mathematical Foundations of Computer Science*, Springer LNCS 45, 576–587 (1976).

A. Ginzburg. *Algebraic Theory of Automata*. Academic Press (1968).

J. Goguen and R. Burstall. Institutions: abstract model theory for specification and programming. *Journal of the ACM* 39:95–146 (1992).

J. Goguen, J. Thatcher and E. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. *Current Trends in Programming Methodology, Vol. 4: Data Structuring* (R.T. Yeh, ed.). Prentice-Hall, 80–149 (1978).

J. Guttag and J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer (1993).

C.A.R. Hoare. Proofs of correctness of data representations. *Acta Informatica* 1:271–281 (1972).

P. Hoffman. Verifying architectural specifications. *Recent Trends in Algebraic Development Techniques, Selected Papers, WADT'01*, Springer LNCS 2267, 152-175 (2001).

F. Honsell, J. Longley, D. Sannella and A. Tarlecki. Constructive data refinement in typed lambda calculus. *Proc. 2nd Intl. Conf. on Foundations of Software Science and Computation Structures*. Springer LNCS 1784, 149–164 (2000).

S. Kahrs, D. Sannella and A. Tarlecki. The definition of Extended ML: a gentle introduction. *Theoretical Comp. Sci.* 173:445–484 (1997).

B. Klin, P. Hoffman, A. Tarlecki, L. Schröder and T. Mossakowski. Checking amalgamability conditions for CASL architectural specifications. *Proc. 26th Intl. Symp. Mathematical Foundations of Computer Science MFCS'01*, Springer LNCS 2136, 451–463 (2001).

R. Milner. An algebraic definition of simulation between programs. *Proc. 2nd Intl. Joint Conf. on Artificial Intelligence*, London, 481–489 (1971).

T. Mossakowski, P. Hoffman, S. Autexier and D. Hutter. CASL Logic. In: (Mosses 2004), 275–359.

P.D. Mosses, ed. CASL *Reference Manual*. Springer LNCS 2960 (IFIP Series), 2004.

A. Popescu and G. Roşu. Behavioral extensions of institutions. *Proc. 1st Conf. on Algebra and Coalgebra in Computer Science CALCO'05*, Swansea. Springer LNCS 3629, 331–347 (2005).

H. Reichel. Behavioural equivalence — a unifying concept for initial and final specification methods. *Proc. 3rd Hungarian Comp. Sci. Conference*, 27–39 (1981).

D. Sannella and A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation* 76:165–210 (1988a).

D. Sannella and A. Tarlecki. Toward formal development of programs from algebraic specifications: implementations revisited. *Acta Informatica* 25:233–281 (1988b).

D. Sannella and A. Tarlecki. Toward formal development of ML programs: foundations and methodology. *Proc. Colloq. on Current Issues in Programming Languages, Intl. Joint Conf. on Theory and Practice of Software Development TAPSOFT'89*, Barcelona. Springer LNCS 352, 375–389 (1989).

D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing* 9:229–269 (1997).

O. Schoett. *Data Abstraction and the Correctness of Modular Programming.* Ph.D. thesis, report CST-42-87, Dept. of Computer Science, Univ. of Edinburgh (1987).

O. Schoett. Behavioural correctness of data representations. *Science of Computer Programming* 14:43–57 (1990).

L. Schröder and T. Mossakowski. HasCasl: Towards integrated specification and development of functional programs. *Proc. 9th Intl. Conf. on Algebraic Methodology and Software Technology, AMAST'02*. Springer LNCS 2422, 99–116 (2002).

L. Schröder, T. Mossakowski, A. Tarlecki, P. Hoffman and B. Klin. Semantics of architectural specifications in Casl. *Proc. 4th Intl. Conf. Fundamental Approaches to Software Engineering FASE'01*, Springer LNCS 2029, 253–268 (2001).

L. Schröder, T. Mossakowski, A. Tarlecki, P. Hoffman and B. Klin. Amalgamation in the semantics of Casl. *Theoretical Computer Science*, 331:215–247 (2005).

C. Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley, New-York, N.Y. (1998).

A. Tarlecki. Abstract specification theory: An overview. In *Models, Algebras and Logic of Engineering Software*, M. Broy and M. Pizka, eds., NATO Science Series - Computer and Systems Sciences, Vol. 191, 43–79, IOS Press, 2003.