Abstract Specification Theory: An Overview

Andrzej TARLECKI*
Institute of Informatics, Warsaw University
and
Institute of Computer Science, Polish Academy of Sciences
Warsaw, Poland

Abstract. This paper presents an overview of abstract specification theory, as understood and viewed by the author. We start with a brief outline of the basic assumptions underlying work in this area in the tradition of algebraic specification and with a sketch of the algebraic and categorical foundations for this work. Then, we discuss the issues of specification construction and of systematic development of software from formal requirements specification. Special attention is paid to architectural design: formal description of the modular structure of the system under development, as captured by architectural specifications in CASL. In particular, we present a simplified but representative formalism of architectural specifications with complete semantics and verification rules. We conclude by adapting the ideas, concepts and results presented to the observational view of software systems and their specification.

1 Introduction

The long-term goal within the area we present here is to provide a *formal basis* for *systematic development* of *correct programs* from *requirements specifications* by *verified refinement steps*. This statement in many variations has been repeated over and over by the author and many others to motivate their work. Indeed: it is full of phrases and key words capturing well the intentions behind many (if not quite all) *formal methods* in software engineering. The stress on the need for precise software specification, correctness of the developed product, proceeding from high-level requirements to the final system in some gradual, systematic manner subject to verification and analysis are the expected and practically important norm, and apply to a wide variety of approaches.

The ideas we present here grew on the tradition of *algebraic specification* [16, 43], although there is by now much more flexibility and power here than in traditional algebraic specifications, dominated by equational specifications with initial algebra semantics. What remained of the purely algebraic tradition of equational algebra, as perhaps first introduced to software specification in its pure form by the ADJ group [21], is primarily the stress on the semantic view of software, with programs modeled essentially as algebras. Indeed: software

*This work has been partially supported by KBN grant 7T11C 002 21 and European AGILE project IST-2001-32747.

systems (programs, modules, databases, ...) can be fruitfully viewed semantically as sets of data with some operations on them — that is, as algebras. Program correctness w.r.t. some formal specification is considered the primary issue, to much extent disregarding, or treating as secondary, the actual code and its quality, readability, efficiency, robustness, reliability, etc. — while all these are, of course, very important, correctness takes a clear precedence over all these other desirable features. *Structuring* specifications, programs, correctness proofs, etc., is viewed as the key practical tool to handle the complexity of software systems and their development tasks that have to be (and are!) undertaken. Unlike in many other approaches, this is not merely an informal methodological directive, but a central point of the approach we present, with precise mathematical foundations aimed at its support and at capturing it in a formal manner.

In fact, the stress on sound mathematical foundations as a necessary part of a truly trustworthy framework for formal specification and systematic development of correct software is perhaps the most predominant feature underlying the approach presented here.

We start by recalling briefly the very basic algebraic and categorical preliminaries, sketching also how the specific concepts underlying traditional algebraic specifications can be abstracted from without sacrificing mathematical rigor and precision (Sect. 2). We discuss then the very concept of a specification (Sect. 3.1) and sketch some of specification mechanisms present in many formalisms (Sect. 3.2) and in CASL [1, 14] in particular (Sect. 3.3). A formal view of the process of software development is presented in Sect. 3.4, leading to the issues of modular design, discussed in Sect. 4 using the concept of architectural specification of CASL [5]. To include some technical novelty, we outline a formal semantics and a proof calculus for a fragment of CASL architectural specifications, extending the work in [6, 39]. Observational interpretation of specifications and its consequences for the concepts underlying our view of software development, including the semantics and proof calculus for architectural specifications, are discussed in Sect. 5. Brief conclusions and comments on further work are in Sect. 6.

Acknowledgments

This paper is an account of my lectures given at Marktoberdorf Summer School 2002 on "Models, Algebras and Logic of Engineering Software". I am grateful to the directors of the school for inviting me there, to the students for stimulating questions and discussions and for helping me to identify the more difficult parts of the lectures, and to the editors of this volume for combining pressure and flexibility to make sure that this paper gets written down.

The ideas, concepts and results presented here have been developed jointly with Don Sannella; in fact, this paper is based on a series of joint papers Don and me have written over nearly 20 years of our collaboration. The work on architectural specifications has been carried out jointly with Michel Bidoit and Don Sannella, with many of the semantic and proof-theoretic aspects developed together with Lutz Schröder, Till Mossakowski, Piotr Hoffman and Bartek Klin. The collaboration within the CoFI group over the last few years have been of invaluable help to crystallize and rephrase many of the key ideas to a more palatable form. Many thanks to them all!

2 Underlying Logical Framework

2.1 Algebraic Preliminaries

A basic intuition on our semantic view of software systems (programs, modules) implies the following rough analogy:

module	$\rightsquigarrow$ algebra
module interface	$\rightsquigarrow$ algebraic signature
module specification	$\rightsquigarrow$ class of algebras

While the "software-engineering" concepts will be left unexplained in this paper, the right column has to be made precise. In fact, these concepts have been a topic of study of universal algebra for years [23], and we can draw on all this work, adapting it slightly to the *many-sorted* (or *heterogeneous* [9]) framework naturally emerging in computer-science applications. We briefly review these concepts here, mainly to fix the notation — referring for instance to [34] for a more complete account.

An *algebraic (many-sorted) signature* $\Sigma = (S, \Omega)$ consists of a set S of *sort names* and of a set $\Omega = \langle \Omega_{w,s} \rangle_{w \in S^*, s \in S}$ of *operation names*, classified by their *arity* and *result sort*. Given such a signature, a Σ-*algebra* $A = (|A|, \langle f_A \rangle_{f \in \Omega})$ consists of a family $|A| = \langle |A|_s \rangle_{s \in S}$ of its *carrier sets* and of *operations* $f_A\colon |A|_{s_1} \times \ldots \times |A|_{s_n} \to |A|_s$, one for each operation name $f \in \Omega_{s_1 \ldots s_n, s}$. We write $f\colon s_1 \times \ldots \times s_n \to s$ for $s_1, \ldots, s_n, s \in S$ and $f \in \Omega_{s_1 \ldots s_n, s}$. The class of all Σ-algebras will be denoted by $Alg(\Sigma)$.

Consider Σ-algebras $A, B \in Alg(\Sigma)$. A *subalgebra* $A_{sub} \subseteq A$ is given by a subset $|A_{sub}| \subseteq |A|$ closed under the operations in A. A *homomorphism* $h\colon A \to B$ is a (many-sorted) map $h\colon |A| \to |B|$ that preserves the operations. A *congruence* ρ *on* A is an equivalence $\rho \subseteq |A| \times |A|$ closed under the operations. The *quotient algebra* A/ρ (by congruence ρ on A) is built in the natural way on the equivalence classes of ρ.

Syntactic concepts and their interpretation are introduced as usual, given an S-sorted set X of variables, Σ-algebra A and valuation $v\colon X \to |A|$. *Terms* $t \in |T_\Sigma(X)|$ are built using variables X and operations (including constants, which are nullary operations) from Ω in the usual way; a *closed* term is a term with no variables. The *term algebra* $T_\Sigma(X)$ has the set of all Σ-terms with variables X as the carrier, and the operations defined "syntactically". The *evaluation* $v^\sharp\colon T_\Sigma(X) \to A$ of Σ-terms under valuation of variables $v\colon X \to |A|$ is the unique homomorphism from $T_\Sigma(X)$ to A that extends v. The *value* of term t in A under v is defined as $t_A[v] = v^\sharp(t)$ (it may also be defined inductively).

Algebraic properties are captured by *equations* of the form $\forall X.t = t'$, where X is a set of variables, and $t, t' \in |T_\Sigma(X)|_s$ are terms of a common sort, via the notion of logical *satisfaction*: a Σ-algebra A *satisfies* such an equation, written $A \models \forall X.t = t'$, if for all $v\colon X \to |A|$, $t_A[v] = t'_A[v]$. This generalizes to sets of equations and classes of algebras in the obvious way. The set of all Σ-equations (with variables taken from a predefined vocabulary) will be denoted by $Eq(\Sigma)$.

For a set $\Phi \subseteq Eq(\Sigma)$ of Σ-equations, the class of its *models* is $Mod(\Phi) = \{A \in Alg(\Sigma) \mid A \models \Phi\}$. For a class $\mathcal{C} \subseteq Alg(\Sigma)$ of Σ-algebras, its *theory* is $Th(\mathcal{C}) = \{\varphi \in Eq(\Sigma) \mid \mathcal{C} \models \varphi\}$. Mod and Th as functions between subsets of $Eq(\Sigma)$ and subclasses of $Alg(\Sigma)$ ordered by inclusion and containment, respectively, form a *Galois connection* (are monotone and satisfy $\mathcal{C} \subseteq Mod(\Phi) \iff Th(\mathcal{C}) \supseteq \Phi$). It follows for instance that the composition $Mod;Th$ is a *closure operator* on sets of sentences; for $\Phi \subseteq Eq(\Sigma)$, $Th(Mod(\Phi))$ is the *theory generated by* Φ. We write $\Phi \models_\Sigma \varphi$ for $\varphi \in Th(Mod(\Phi))$ (omitting the subscript Σ if the signature is evident from the context). In other words: $\Phi \models \varphi$ if φ is a *semantic consequence* of Φ, that is, φ holds in every model of Φ.

A point to notice here is that in the many-sorted framework (unlike in the traditional, single-sorted algebra) one is forced to pay a special attention to the set of variables over which the equations are considered [20]. For instance, over a signature with constants a and b and a unary operation f, $a = b$ is *not* a consequence of $a = f(x)$ and $f(x) = b$ (unless there is a closed term of the argument sort of f).

Semantic consequence is the primary target for the development of reasoning mechanisms to deduce consequences of a set of equations without referring directly to the semantic definition. This is achieved by the *equational calculus*, as given by the rules of inference in Fig. 1. A Σ-equation φ is a *proof-theoretic consequence* of a set of Σ-equations Φ, written $\Phi \vdash_\Sigma \varphi$ (with Σ omitted when evident), if φ can be derived from Φ by the rules in Fig. 1.

$$\frac{}{\forall X.t = t} \qquad \frac{\forall X.t = t'}{\forall X.t' = t} \qquad \frac{\forall X.t = t' \quad \forall X.t' = t''}{\forall X.t = t''}$$

$$\frac{\forall X.t_1 = t'_1 \quad \ldots \quad \forall X.t_n = t'_n}{\forall X.f(t_1 \ldots t_n) = f(t'_1 \ldots t'_n)} \qquad \frac{\forall X.t = t'}{\forall Y.t_{T_\Sigma(Y)}[\theta] = t'_{T_\Sigma(Y)}[\theta]} \text{ for } \theta\colon X \to |T_\Sigma(Y)|$$

Figure 1: The rules of equational calculus

Of course, a calculus like this requires some justification. In this case the situation is as good as one can expect: only "true" facts can be derived in the equational calculus (this is referred to as *soundness* of the calculus), and moreover, all such facts can be derived (*completeness* of the calculus). The semantic and proof-theoretic consequences coincide:

Fact 1. *The equational calculus is sound and complete: for any $\Phi \subseteq Eq(\Sigma)$ and $\varphi \in Eq(\Sigma)$,*

$$\Phi \models \varphi \iff \Phi \vdash \varphi.$$

Most of traditional universal algebra (and logic) has been done for an "arbitrary but fixed" signature, with notable exception for the study of so-called theory morphisms, which in applications in computer science play a much more prominent role. The possibility to move from one signature to another (to add, change, remove or glue together various signature components) is of crucial importance at various stages of program specification and development. This capability is introduced by the notion of a signature morphism.

Given signatures $\Sigma = (S, \Omega)$ and $\Sigma' = (S', \Omega')$, a *signature morphism* $\sigma\colon \Sigma \to \Sigma'$ maps sort names to sort names, $\sigma\colon S \to S'$, and operation names to operation names, preserving their profiles: $\sigma\colon \Omega_{w,s} \to \Omega'_{\sigma(w),\sigma(s)}$, for $w \in S^*$, $s \in S$.

Such a signature morphism $\sigma\colon \Sigma \to \Sigma'$ determines in a natural way the translation of "Σ-syntax" to "Σ'-syntax":

- *translation of variables*: given an S-sorted set X of variables, σ maps it to an S'-sorted set X', where $X'_{s'} = \biguplus_{\sigma(s)=s'} X_s$ (with $\biguplus$ standing for disjoint union of sets);
- *translation of terms*: $\sigma\colon |T_\Sigma(X)|_s \to |T_{\Sigma'}(X')|_{\sigma(s)}$, for $s \in S$, defined by substituting $\sigma(f)$ for f, for each operation name f in Σ, and re-sorting the variables as given in their translation;
- *translation of equations*: $\sigma\colon Eq(\Sigma) \to Eq(\Sigma')$, with $\sigma(\forall X.t_1 = t_2)$ yielding $\forall X'.\sigma(t_1) = \sigma(t_2)$.

Perhaps less expected, but equally simple and natural, is that a signature morphism $\sigma\colon \Sigma \to \Sigma'$ determines a translation of "Σ'-semantics" to "Σ-semantics" (note the contravariancy!):

- *σ-reduct*: $\_|_\sigma\colon Alg(\Sigma') \to Alg(\Sigma)$, where for $A' \in Alg(\Sigma')$, $|A'|_\sigma|_s = |A'|_{\sigma(s)}$ for $s \in S$, and $f_{A'|_\sigma} = \sigma(f)_{A'}$ for $f \in \Omega$.

The two translations are linked by the following crucial fact, known as the *satisfaction property*:

Fact 2. *For any signature morphism $\sigma\colon \Sigma \to \Sigma'$, Σ'-algebra A' and Σ-equation φ:*

$$A'\big|_\sigma \models_\Sigma \varphi \iff A' \models_{\Sigma'} \sigma(\varphi).$$

This captures the key properties of satisfaction: very informally, Fact 2 states that "truth" is preserved under change of notation (names for the operations used) and restriction/extension of irrelevant context (semantic interpretation in algebras of the symbols not used in the equation).

2.2 Equational Specifications

Recalling that the intention is to model programs (software modules) as algebras, the *equational logic* as introduced in Sect. 2.1 offers an obvious way to build specifications: just indicate a signature to determine the static module interface and a set of axioms (equations over this signature) to determine the required module properties. A module is *correct* w.r.t. such a specification, if the algebra it semantically denotes satisfies the axioms.

An *equational specification* $\langle \Sigma, \Phi \rangle$ consists of a signature Σ and a set $\Phi \subseteq \mathit{Eq}(\Sigma)$ of Σ-equations. It determines its *signature* $\mathit{Sig}[\langle \Sigma, \Phi \rangle] = \Sigma$ and the class of its *models* $\mathit{Mod}[\langle \Sigma, \Phi \rangle] = \mathit{Mod}(\Phi)$.

Here is a simple example, in a hopefully self-evident notation taken from CASL [1, 14]:

spec NAIVENAT = **sort** Nat
ops $0 : Nat;$
$succ : Nat \to Nat;$
$\_+\_ : Nat \times Nat \to Nat$
axioms $\forall n{:}Nat \bullet n + 0 = n;$
$\forall n, m{:}Nat \bullet n + succ(m) = succ(n + m)$

The following characterization describes precisely the classes of algebras that equational specifications may define:

Fact 3. *A class of Σ-algebras is equationally definable iff it is closed under subalgebras, products and homomorphic images.*

This means in particular that no matter how cleverly one uses equational logic, only some classes of models can be described by equational specifications. Consequently, the trouble is that equational specifications typically admit a lot of undesirable models.

For instance, NAIVENAT admits a trivial model, where there is only one element, and so all the equations hold, as well as models where $+$ is not commutative[1]:

$$\text{NAIVENAT} \not\models \forall n, m{:}Nat \bullet n + m = m + n.$$

The natural proof of commutativity for $+$ relies on induction, not available (since not sound in arbitrary algebras) for the equational calculus. This indicates that one way to alleviate the shortcomings of equational specifications is to additionally equip them with extra-logical *constraints* (the terminology originates from the work on CLEAR [12]):

- *reachability constraint*: imposes "no junk" condition, which requires that all carrier elements in algebras considered are values of closed terms. This is further generalized to *generation constraints*, permitting some sorts to be considered as unconstrained "input".

[1] For instance, consider countable, possibly transfinite sequences of ticks, where 0 is the empty sequence, $succ$ adds a tick at the end of its argument, even if it is infinite, and $+$ is concatenation.

- *initiality constraint*: apart from "no junk" imposes "no confusion" condition, which requires that no values of two terms are identified unless this follows from the axioms. This is further generalized to *freeness constraints*, permitting reducts of the algebra to a subsignature to be considered as unconstrained "input".

Using CASL notation again to impose the initiality constraint on our specification, we can introduce a "better" specification of the module for natural numbers:

$$
\begin{array}{l}
\textbf{spec } \text{NAT} = \textbf{free } \{ \ \textbf{sort } Nat \\
\qquad\qquad\qquad\qquad \textbf{ops } 0 : Nat; \\
\qquad\qquad\qquad\qquad\qquad succ : Nat \to Nat; \\
\qquad\qquad\qquad\qquad\qquad \_+\_ : Nat \times Nat \to Nat \\
\qquad\qquad\qquad\qquad \textbf{axioms } \forall n{:}Nat \bullet n + 0 = n; \\
\qquad\qquad\qquad\qquad\qquad\quad \forall n, m{:}Nat \bullet n + succ(m) = succ(n + m) \\
\qquad\qquad\qquad\quad \}
\end{array}
$$

Now, this specification selects only initial models of NAIVENAT (satisfying "no junk" and "no confusion" requirements), which have countably many distinct elements denoted by 0, $succ(0)$, $succ(succ(0))$, etc., and therefore:

$$\text{NAT} \models \forall n, m{:}Nat \bullet n + m = m + n.$$

In fact, in CASL initiality constraints need not encompass the entire specification, and may be put exactly where they belong. The above specification may be replaced by perhaps a more natural one, imposing the initiality constraint on the declaration of the data type Nat with its constructors only, without modifying the overall semantics of NAT:

$$
\begin{array}{l}
\textbf{spec } \text{NAT}' = \textbf{free type } Nat ::= 0 \mid succ(Nat) \\
\qquad\qquad\quad \textbf{op } \_+\_ : Nat \times Nat \to Nat \\
\qquad\qquad\quad \textbf{axioms } \forall n{:}Nat \bullet n + 0 = n; \\
\qquad\qquad\qquad\qquad \forall n, m{:}Nat \bullet n + succ(m) = succ(n + m)
\end{array}
$$

Generation constraints may be used similarly. A warning: when initiality or reachability constraints are permitted, there can be no sound and complete finitary proof system for deriving consequences of such specifications [25].

Adding constraints to our specifications in fact means that we go beyond equational logic to increase the definitional power of our specification. This is just one example: other logical systems of quite diverse power have been proposed and used in specifications. To cope with the resulting "population explosion" among logical systems used in formal specifications we need machinery even more abstract than universal algebra.

2.3 Categorical Preliminaries

Category theory is a branch of mathematics where, very informally, objects of interest are considered without looking into their internal structure but rather through their relationship with other objects. It has its own deep developments, interesting interactions with other areas of mathematics, and important applications in other domains of science. Fortunately, we will not need to rely on, nor develop ourselves, any deep categorical concepts and results. We just use the basics of category theory, essentially taking its vocabulary of abstract concepts as a convenient language to deal with the ideas of software specification and development at a sufficiently general level. In this section we briefly review the concepts needed, largely to fix our notation and terminology, referring to many of the standard textbooks and monographs on category theory for a fully formal presentation (with classical [24] still a primary reference; we add [35], with a presentation targeted at exactly the area we address here).

Let us start with the most basic concepts. A *category* $\mathbf{K}$ consists of:

- a class of *objects*: $|\mathbf{K}|$; the category is *small* if this is a set;
- sets of *morphisms*: $\mathbf{K}(A, B)$, for all $A, B \in |\mathbf{K}|$; $m\colon A \to B$ stands for $m \in \mathbf{K}(A, B)$;
- morphism *composition*: for $m\colon A \to B$ and $m'\colon B \to C$, we have $m; m'\colon A \to C$; the composition is associative and has identities.

A *functor* $\mathbf{F}\colon \mathbf{K} \to \mathbf{K}'$ between categories maps $\mathbf{K}$-objects to $\mathbf{K}'$-objects, and $\mathbf{K}$-morphisms to $\mathbf{K}'$-morphisms, preserving their source and target, composition and identities.

For each category $\mathbf{K}$, there is the *opposite* (*dual*) category $\mathbf{K}^{op}$, with the same objects and morphisms, but with the direction of morphisms formally reversed. Every categorical concept and fact has its dual: the same concept or fact considered in a dual category. A *contravariant* functor from $\mathbf{K}$ to $\mathbf{K}'$ is a functor as defined above from $\mathbf{K}^{op}$ to $\mathbf{K}'$.

We have already encountered in this paper a number of categories and functors (we do not spell out below the natural morphism compositions):

- the category $\mathbf{Set}$ of sets, with sets as objects and functions as morphisms;
- for each signature Σ, the category $\mathbf{Alg}(\Sigma)$ with Σ-algebras as objects and their homomorphisms as morphisms;
- the category $\mathbf{AlgSig}$ of algebraic signatures and their morphisms;
- the category $\mathbf{Cat}$ of (sm)all categories and functors between them;
- for each signature morphism $\sigma\colon \Sigma \to \Sigma'$, σ-reduct extends to the functor $\_|_\sigma\colon \mathbf{Alg}(\Sigma') \to \mathbf{Alg}(\Sigma)$ (extracting a Σ-homomorphism out of any Σ'-homomorphism as expected);
- $\mathbf{Eq}\colon \mathbf{AlgSig} \to \mathbf{Set}$ is a (covariant) functor mapping each signature Σ to the set $Eq(\Sigma)$ and signature morphism $\sigma\colon \Sigma \to \Sigma'$ to the translation function $\sigma\colon Eq(\Sigma) \to Eq(\Sigma')$;
- $\mathbf{Alg}\colon \mathbf{AlgSig}^{op} \to \mathbf{Cat}$ is a (contravariant) functor mapping each signature Σ to the category $\mathbf{Alg}(\Sigma)$ and morphism $\sigma\colon \Sigma \to \Sigma'$ to the reduct functor $\_|_\sigma\colon \mathbf{Alg}(\Sigma') \to \mathbf{Alg}(\Sigma)$.

If J is a small category, a functor $D\colon J \to \mathbf{K}$ may be thought of as a *diagram* in $\mathbf{K}$ of the *shape* J; equivalently, a diagram in $\mathbf{K}$ consists of a directed graph with nodes labeled by $\mathbf{K}$-objects and edges by $\mathbf{K}$-morphisms so that their source and target are preserved.

Given such a diagram, an important concept is that of a *limit* and its dual, a *colimit*. Rather than defining them in full generality (see e.g. [24, 35]) let's look at some special cases. A *pullback* is a limit of a diagram that consists of two morphisms with a common target, say, $f\colon A \to C$ and $g\colon B \to C$. Then a pullback of f and g is an object $lim(f, g)$ with morphisms $f'\colon lim(f, g) \to B$ and $g'\colon lim(f, g) \to A$ such that $f'; g = g'; f$ and for any object P with morphisms $f''\colon P \to B$ and $g''\colon P \to A$ such that $f''; g = g''; f$, there exists a unique morphism $j\colon P \to lim(f, g)$ such that $j; g' = g''$ and $j; f' = f''$ (see Fig. 2).

Figure 3 names a few frequently encountered limits and suggests their construction in the category $\mathbf{Set}$ of sets and functions between them (π_A, π_B are projections from the Cartesian product, "hooked" arrow indicates inclusion).

The intuition here should be that a diagram lists objects with morphisms indicating how one object is built over another; then the limit collects all tuples of consistent elements of the objects involved in the diagram. This is suggested by examining the construction of products and equalizers in $\mathbf{Set}$. In a way, we do not need to look at the construction of other limits:

$$
\begin{tikzcd}
 & C & \\
A \arrow[ur, "f"] & \textsf{PB} & B \arrow[ul, "g"'] \\
 & lim(f,g) \arrow[ul, "g'"] \arrow[ur, "f'"'] & \\
 & P \arrow[uul, "g''"] \arrow[uur, "f''"'] \arrow[u, "j"'] &
\end{tikzcd}
\qquad
\begin{tikzcd}
 & C \arrow[dl, "f"'] \arrow[dr, "g"] & \\
A \arrow[dr, "g'"] \arrow[ddr, "g''"'] & \textsf{PO} & B \arrow[dl, "f'"'] \arrow[ddl, "f''"] \\
 & colim(f,g) \arrow[d, "j"] & \\
 & P &
\end{tikzcd}
$$

Figure 2: Pullback and pushout

Fact 4. *All limits can be constructed using products (of arbitrary families of objects) and equalizers.*

In particular, it is worth expressing the pullback as composition of a product and an equalizer; the construction indicated in Fig. 3 follows from this, and agrees with the intuition for limits suggested above. In a sense, the converse of this construction is also possible: binary products and equalizers (and hence all finite limits) can be constructed using pullbacks and a terminal object.

diagram	limit	in $\mathbf{Set}$
(empty)	*terminal object*	$\{*\}$ (any singleton set)
$A \qquad B$	*product*	$A \xleftarrow{\pi_A} A \times B \xrightarrow{\pi_B} B$
$A \overset{f}{\underset{g}{\rightrightarrows}} B$	*equalizer*	$\{a \in A \mid f(a) = g(a)\} \hookrightarrow A$
$A \xrightarrow{f} C \xleftarrow{g} B$	*pullback*	$\{(a,b) \in A \times B \mid f(a) = g(b)\}$ with the obvious projections to A and B

Figure 3: Some limits in $\mathbf{Set}$

The notion dual to a limit is that of a *colimit* of a diagram. In particular, the notion dual to a pullback is a *pushout* of morphisms with a common source; it is a good exercise to define this explicitly, by dualizing the definition given for a pullback (see Fig. 2). Figure 4 names a few frequently encountered colimits and suggests their construction in $\mathbf{Set}$ (ι_A, ι_B are injections into disjoint union). The intuition is now that a diagram lists objects with morphisms indicating how one object extends another; the colimit adds the elements of all objects up, gluing together the elements originating from a common source. All colimits may be constructed by using coproducts (of arbitrary families of objects) and coequalizers (by a dual to Fact 4). So, to justify this intuition, it is enough to look at coproducts and coequalizers, and perhaps to check this out by building a pushout as a coproduct followed by a coequalizer.

diagram	colimit	in $\mathbf{Set}$
(empty)	*initial object*	$\emptyset$
$A \qquad B$	*coproduct*	$A \xrightarrow{\iota_A} A \uplus B \xleftarrow{\iota_B} B$
$A \overset{f}{\underset{g}{\rightrightarrows}} B$	*coequalizer*	$B \longrightarrow B/{\equiv}$ where $f(a) \equiv g(a)$ for all $a \in A$
$A \xleftarrow{f} C \xrightarrow{g} B$	*pushout*	$A \xrightarrow{g'} (A \uplus B)/{\equiv} \xleftarrow{f'} B$ where $\iota_A(f(c)) \equiv \iota_B(g(c))$ for all $c \in C$, with $g'(a) = [\iota_A(a)]_\equiv$ and $f'(b) = [\iota_B(b)]_\equiv$

Figure 4: Some colimits in $\mathbf{Set}$

Again dually to the case for limits, we can construct binary coproducts and coequalizers (and hence all finite colimits) using pushouts and an initial object.

The constructions of limits and colimits in $\mathbf{Set}$ are quite typical: with relatively minor and often self-evident modifications, they work in many other categories. For instance, Fig. 5 shows a pushout in the category $\mathbf{AlgSig}$ of algebraic signatures.

$$\begin{tikzcd}
\boxed{\begin{array}{l}\textbf{sort } \mathit{Nucl}\\ \textbf{ops } A, C, G, T : \mathit{Nucl}\end{array}} \arrow[r] & \boxed{\begin{array}{l}\textbf{sorts } \mathit{Nucl}, \mathit{Nat}, \mathit{Array}\\ \textbf{ops } A, C, G, T : \mathit{Nucl};\\ \mathit{empty} : \mathit{Array}; \mathit{size} : \mathit{Array} \to \mathit{Nat};\\ \mathit{get} : \mathit{Nat} \times \mathit{Array} \to \mathit{Nucl};\\ \mathit{put} : \mathit{Nat} \times \mathit{Nucl} \times \mathit{Array} \to \mathit{Array}\end{array}} \\
\boxed{\textbf{sort } \mathit{Elem}} \arrow[u] \arrow[r] \arrow[ur, phantom, "\text{PO}"] & \boxed{\begin{array}{l}\textbf{sorts } \mathit{Elem}, \mathit{Nat}, \mathit{Array}\\ \textbf{ops } \mathit{empty} : \mathit{Array}; \mathit{size} : \mathit{Array} \to \mathit{Nat};\\ \mathit{get} : \mathit{Nat} \times \mathit{Array} \to \mathit{Elem};\\ \mathit{put} : \mathit{Nat} \times \mathit{Elem} \times \mathit{Array} \to \mathit{Array}\end{array}} \arrow[u]
\end{tikzcd}$$

Figure 5: A pushout of signatures in $\mathbf{AlgSig}$

A category with limits (resp., colimits) of all (finite) diagrams is called (finitely) *complete* (resp., *cocomplete*). Of course, there are categories where some limits and some colimits do not exist. If they exist, they are always defined up to an isomorphism. It follows that the constructions sketched above for limits and colimits in $\mathbf{Set}$ may be used to identify the result up to isomorphism only. As often in category theory (and universal algebra), we will gloss over the differences between distinct but isomorphic objects.

2.4 *Institutions*

We have already indicated in Sect. 2.2 the need for modification of the underlying logical system so that formal specifications may better capture required properties of algebras. The corresponding tuning up of logic considered in the literature and used in practice involved at least:

- various sets of formulae (Horn-clauses, first-order, higher-order, modal formulae, ...);

- various notions of algebra (partial algebras, relational structures, error algebras, Kripke structures, ...);
- various notions of signature (order-sorted, error, higher-order signatures, sets of propositional variables, ...);
- various notions of signature morphisms, with various translations of formulae and reducts of algebras.

The resulting "population explosion" among logical systems used in software specification has not and could not lead to any specific "best" system, suitable for every need that may arise in the theory and practice of software specification. Therefore, following Goguen and Burstall [19], rather than engaging in a futile search for a unique ideal system, we propose to abstract away from the details of whatever logical system one may want to use, and develop at least the basic concepts, methods and results working within a system we assume as little as possible about, thus making the developments applicable to an essentially arbitrary logical system. To realize this plan, one needs a formal concept of a what a logical system is. This has been provided in a form adequate for the purposes of abstract specification theory (and not only for this) via the notion of institution, formed very much in the spirit of *abstract model theory* [2, 3]. We refer to the original paper on institutions [19] and to [42, 36] for a more exhaustive presentation of this concept and the related theory, limiting the presentation here to the basic definition, a list of a few examples, and an abstract formulation of some logical properties in this setting.

An *institution* consists of: a category $\mathbf{Sign}$ of *signatures*, functors $\mathbf{Sen}\colon \mathbf{Sign} \to \mathbf{Set}$ and $\mathbf{Mod}\colon \mathbf{Sign}^{op} \to \mathbf{Cat}$, and for each $\Sigma \in |\mathbf{Sign}|$, a Σ-*satisfaction relation* $\models_\Sigma \subseteq |\mathbf{Mod}(\Sigma)| \times \mathbf{Sen}(\Sigma)$. For $\Sigma \in |\mathbf{Sign}|$, $\mathbf{Sen}(\Sigma)$ is the set of Σ-*sentences*, and $\mathbf{Mod}(\Sigma)$ is the category of Σ-*models* and their morphisms. For any signature morphism $\sigma\colon \Sigma \to \Sigma'$ in $\mathbf{Sign}$, $\mathbf{Sen}(\sigma)$, written as $\sigma\colon \mathbf{Sen}(\Sigma) \to \mathbf{Sen}(\Sigma')$, is a translation of sentences, and $\mathbf{Mod}(\sigma)$, written as $\_|_\sigma\colon \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$ is a *reduct functor*. These are subject to the following *satisfaction condition*:

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

where $\sigma\colon \Sigma \to \Sigma'$ in $\mathbf{Sign}$, $M' \in |\mathbf{Mod}(\Sigma')|$, $\varphi \in \mathbf{Sen}(\Sigma)$.

The requirements this definition imposes on a logical system are very mild, and in fact all typical logics may be put into this mold. Here are some typical examples of institutions:

- $\mathbf{EQ}$ — equational logic;
- $\mathbf{FOEQ}$ — first-order logic (with predicates and equality);
- $\mathbf{PEQ}$, $\mathbf{PFOEQ}$ — as above, but with partial operations;
- $\mathbf{HOL}$ — higher-order logic (say, in the HOL version, [22]);
- logics of constraints (fitted via signature morphisms);
- $\mathbf{CASL}$ — the logic of CASL: partial first-order logic with equality, predicates, generation constraints, and subsorting.

Just to show how much can be covered, let us mention that modal logics, many-valued logics, and even programming language semantics may be formalized as institutions. Also, many perhaps unexpected institutions may be defined, e.g. with: no sentences, no models, no signatures, trivial satisfaction relations, sets of sentences as sentences, sets of sentences as signatures, classes of models as sentences, sets of sentences as models, etc.

For the rest of this section, consider an institution $\mathbf{I} = (\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \langle \models_\Sigma \rangle_{\Sigma \in |\mathbf{Sign}|})$.

For any signature $\Sigma \in |\mathbf{Sign}|$, the satisfaction relation between Σ-models and Σ-sentences allows us to repeat many concepts and results of the equational case as presented in Sect. 2.1. In particular, the definitions of the class of *models*[2] $Mod(\Phi) \subseteq |\mathbf{Mod}(\Sigma)|$ of a set of sentences $\Phi \subseteq \mathbf{Sen}(\Sigma)$, and of the *theory* $Th(\mathcal{M}) \subseteq \mathbf{Sen}(\Sigma)$ of a class of models $\mathcal{M} \subseteq |\mathbf{Mod}(\Sigma)|$, carry over without change from there. As before, Mod and Th form a Galois connection, and determine a closure operator $Mod;Th$ on sets of sentences. The concept of *semantic consequence* carries over as well: for $\Phi \subseteq \mathbf{Sen}(\Sigma)$ and $\varphi \in \mathbf{Sen}(\Sigma)$, $\Phi \models_\Sigma \varphi$ if φ holds in all models of Φ (that is, $\varphi \in Th(Mod(\Phi))$).

One important general fact that holds for an arbitrary institution (since it follows from the satisfaction condition) is the preservation of semantic consequence under translation along signature morphisms:

Fact 5. *For* $\sigma\colon \Sigma \to \Sigma'$, $\Phi \subseteq \mathbf{Sen}(\Sigma)$, *and* $\varphi \in \mathbf{Sen}(\Sigma)$, $\Phi \models_\Sigma \varphi \implies \sigma(\Phi) \models_{\Sigma'} \sigma(\varphi)$. *Moreover, if the reduct* $\_|_\sigma\colon |\mathbf{Mod}(\Sigma')| \to |\mathbf{Mod}(\Sigma)|$ *is surjective, then*

$$\Phi \models_\Sigma \varphi \iff \sigma(\Phi) \models_{\Sigma'} \sigma(\varphi).$$

Of course, we cannot expect that some proof calculus is developed for an entirely arbitrary institution: this clearly has to depend on the specific logic one deals with. However, we can accommodate proof-theoretic considerations by assuming that the institution we consider comes additionally equipped with *proof-theoretic entailment*, given as a family of relations between, for each signature Σ, sets of Σ-sentences and Σ-sentences:

$$\vdash_\Sigma \subseteq \wp(\mathbf{Sen}(\Sigma)) \times \mathbf{Sen}(\Sigma)$$

For such proof-theoretic entailment we assume closure under weakening, reflexivity, transitivity (cut), and under translation along signature morphisms. We also assume the entailment to be *sound*: $\Phi \vdash_\Sigma \varphi \implies \Phi \models_\Sigma \varphi$, and discuss its *completeness*: $\Phi \models_\Sigma \varphi \implies \Phi \vdash_\Sigma \varphi$, for $\Phi \subseteq \mathbf{Sen}(\Sigma)$ and $\varphi \in \mathbf{Sen}(\Sigma)$. The resulting extension of the notion of institution has been named *general logic* in [26].

With the intuition inherited from Sect. 2.3, it should be clear that we may use colimits (in particular, pushouts) in $\mathbf{Sign}$ to combine signatures. Interestingly, colimits at the level of signatures can be lifted to theories (and to specifications); see [19, 42] and Fact 10 for a precise statement. Perhaps more intriguing is the problem whether each combination of signatures is reflected by the corresponding combination of models. This is known as the *amalgamation property*. We say that an institution *admits amalgamation* if for any pushout in $\mathbf{Sign}$

$$\begin{tikzcd}
& \Sigma' & \\
\Sigma_1 \arrow[ur, "\sigma'_2"] & \mathsf{PO} & \Sigma_2 \arrow[ul, "\sigma'_1"'] \\
& \Sigma \arrow[ul, "\sigma_1"] \arrow[ur, "\sigma_2"'] &
\end{tikzcd}$$

given $M_1 \in |\mathbf{Mod}(\Sigma_1)|$ and $M_2 \in |\mathbf{Mod}(\Sigma_2)|$ with $M_1|_{\sigma_1} = M_2|_{\sigma_2}$, there is a unique $M' \in |\mathbf{Mod}(\Sigma')|$ with $M'|_{\sigma'_1} = M_2$ and $M'|_{\sigma'_2} = M_1$ (see Fig. 6). We will see how amalgamation is used to capture consistent module combination in Sect. 3.4.

The amalgamation property follows from the construction of limits in $\mathbf{Cat}$ if we assume that the model functor $\mathbf{Mod}\colon \mathbf{Sign}^{op} \to \mathbf{Cat}$ is *continuous*, that is, maps colimits of signatures to limits of model categories. In particular, the amalgamation property holds for a

[2]We apologize for the traditional overloading of the term "model" here.

$$
\begin{tikzcd}
 & & M' \arrow[dll] \arrow[drr] & & \\
 & & \Sigma' & & \\
M_1 = M'\big|_{\sigma'_2} \arrow[drr] & \Sigma_1 \arrow[ur, "\sigma'_2"] & \textsf{PO} & \Sigma_2 \arrow[ul, "\sigma'_1"'] & M'\big|_{\sigma'_1} = M_2 \arrow[dll] \\
 & & \Sigma \arrow[ul, "\sigma_1"] \arrow[ur, "\sigma_2"'] & & \\
 & & M_1\big|_{\sigma_1} = M_2\big|_{\sigma_2} & &
\end{tikzcd}
$$

Figure 6: Amalgamation property

pushout in $\mathbf{Sign}$ if $\mathbf{Mod}$ maps it to a pullback in $\mathbf{Cat}$, see Fig. 7. It is easy to check that for instance the model functor $\mathbf{Alg}\colon \mathbf{AlgSig}^{op} \to \mathbf{Cat}$ of the equational institution $\mathbf{EQ}$ has this property. This also holds for other standard many-sorted institutions, but continuity of the model functor often fails for their single-sorted versions, where pushouts of signatures are typically mapped to pullbacks of categories, but products are not preserved (initial single-sorted signatures are not in general mapped to terminal single-model categories). Among the institutions mentioned above, $\mathbf{CASL}$, the institution of CASL logic with subsorting, does not admit amalgamation in general; see [39] for the analysis of and ways around this problem.

$$
\begin{tikzcd}
\Sigma_1 \arrow[r, "\sigma'_2"] & \Sigma' & & \mathbf{Mod}(\Sigma_1) \arrow[d, "-|_{\sigma_1}"'] \arrow[dr, phantom, "\textsf{PB}"] & \mathbf{Mod}(\Sigma') \arrow[l, "-|_{\sigma'_2}"'] \arrow[d, "-|_{\sigma'_1}"] \\
\Sigma \arrow[u, "\sigma_1"] \arrow[r, "\sigma_2"'] \arrow[ur, phantom, "\textsf{PO}"] & \Sigma_2 \arrow[u, "\sigma'_1"'] \arrow[r, phantom, "\stackrel{\mathbf{Mod}}{\longmapsto}"] & & \mathbf{Mod}(\Sigma) & \mathbf{Mod}(\Sigma_2) \arrow[l, "-|_{\sigma_2}"]
\end{tikzcd}
$$

Figure 7: Mapping signature pushout to pullback of model categories

Another important property relevant here is *interpolation*: an institution has interpolation if for any pushout in $\mathbf{Sign}$ as above, for all $\varphi_1 \in \mathbf{Sen}(\Sigma_1)$ and $\varphi_2 \in \mathbf{Sen}(\Sigma_2)$ such that $\sigma'_2(\varphi_1) \models_{\Sigma'} \sigma'_1(\varphi_2)$, there is $\theta \in \mathbf{Sen}(\Sigma)$ with $\varphi_1 \models_{\Sigma_1} \sigma_1(\theta)$ and $\sigma_2(\theta) \models_{\Sigma_2} \varphi_2$ (see Fig. 8).

$$
\begin{tikzcd}
 & & \sigma'_2(\varphi_1) \models_{\Sigma'} \sigma'_1(\varphi_2) & & \\
 & & \Sigma' & & \\
\varphi_1 \models_{\Sigma_1} \sigma_1(\theta) \arrow[urr] & \Sigma_1 \arrow[ur, "\sigma'_2"] & \textsf{PO} & \Sigma_2 \arrow[ul, "\sigma'_1"'] & \sigma_2(\theta) \models_{\Sigma_2} \varphi_2 \arrow[ull] \\
 & & \Sigma \arrow[ul, "\sigma_1"] \arrow[ur, "\sigma_2"'] & & \\
 & & \theta \arrow[ull] \arrow[urr] & &
\end{tikzcd}
$$

Figure 8: Interpolation property

Interpolation is a very subtle and deep property. It is well-known that the standard (single-sorted) first-order logic has interpolation [13]; this carries over to the many-sorted case, but only for signature morphisms that are injective on sort names [10]. On the other hand, the equational logic $\mathbf{EQ}$ does not have interpolation as presented here; for signature morphisms injective on sort names, $\mathbf{EQ}$ has interpolation in a version that allows sets of interpolants (instead of a single interpolant θ as above).

3 Specifications and Program Development

3.1 Methodological and Semantic Preliminaries

Following the ideas of Sect. 2.4, throughout this section we will work again in the context of an arbitrary institution $\mathbf{I} = (\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \langle \models_\Sigma \rangle_{\Sigma \in |\mathbf{Sign}|})$, perhaps imposing extra requirements on its structure and properties, but only when they become really necessary.

The informal assumption is, of course, that the institution has been chosen so that the models reflect well the semantic features of the programs (modules, software components) we aim at. Consequently, the rough analogy of Sect. 2.1 takes a revised, more general form:

module	$\rightsquigarrow$ model
module interface	$\rightsquigarrow$ signature
module specification	$\rightsquigarrow$ class of models

A specification is often viewed as a contract between program user and program developer. For the former, it presents all the properties of the program the user can rely on. For the latter, it captures all the requirements the program must fulfill. Needless to add, it is the same specification that must serve both purposes, and just as any other contract, it should be fixed before the actual work begins. Of course, modifications, annexes and various adjustments are possible later on, but both sides must realize that these should be kept to a minimum, since later major changes to the specification contract may turn out extremely costly.

With only a minor twist, we think of a specification as an interface between the actual module realization and its user — either the end system user, or the higher-level system components that use the module covered by the specification. By allowing the user to rely only on the module properties as captured in the specification, it limits the information flow and mutual dependencies between system components (and between their development teams), thus allowing the unavoidable changes and backtracking in system development to be kept as local (and therefore as cheap) as possible. The key property is that a specific module realization may be safely replaced by a new version as long as the latter also satisfies the specification. If this cannot be ensured, modular structure of the system with specifications controlling dependencies between various modules should help to keep the necessary changes as local as possible and to maintain overall control over development.

We will work here with formal specifications which are given a precise, mathematical meaning. In this way both the specifications and the correctness of modules w.r.t. them become amenable to formal analysis.

We will not dwell on the actual syntax of our specifications. There have been quite a number of *specifications languages* designed, starting with CLEAR [12] — see [37] for an overview. CASL [1, 14] is a recent development in this area; we have already used it in examples, and will continue doing so (with a bit more explanation in Sect. 3.3 below). What matters for this paper is that specifications come with a well-defined semantics. For each specification SP, this is given by its *signature*, $Sig[SP] \in |\mathbf{Sign}|$, and the class of its *models*, $Mod[SP] \subseteq |\mathbf{Mod}(Sig[SP])|$.

Two specifications SP_1 and SP_2 are *equivalent*, written $SP_1 \equiv SP_2$, if they have the same semantics, that is, $Sig[SP_1] = Sig[SP_2]$ and $Mod[SP_1] = Mod[SP_2]$. For instance, referring to examples in Sect. 2.2, NAT $\equiv$ NAT$'$.

The semantics of specifications determines their semantic consequences: given a specification SP, a sentence $\varphi \in \mathbf{Sen}(Sig[SP])$ is a *semantic consequence* of SP, written $SP \models \varphi$, if φ holds in every model of SP (that is, $\varphi \in Th(Mod[SP])$). This further extends to consequence between specifications (over the same signature): given specifications SP and SP' with $Sig[SP] = Sig[SP']$, SP' is a *semantic consequence* of SP (or SP is *stronger than* SP'), written $SP \models SP'$, if $Mod[SP] \subseteq Mod[SP']$.

How to build "good" specifications? This is the domain of *specification engineering*, see e.g. [15]. Specification development consists in establishing the desirable system (module) properties and then designing a specification to capture them. This is never an easy task; it typically implies formalization of so far informal user expectations. Various *specification validation* techniques then come in to support checking whether the specification indeed captures all and only desired system properties. All this is very important, but rarely, and if so then to a small extent only, subject to precise formalization. We leave this largely out of the scope of this paper.

Let us mention though that checking consequences of a specification provides the basic means for specification validation, perhaps more fundamental than other techniques, including prototyping and testing. In a way, the latter is but a special case of checking specification consequences: in the simplest case, instead of performing a test on a specification by evaluating in some prototype a term t expecting it to yield a value v, one may "simply" check that $t = v$ is a semantic consequence of the specification. Of course, some proof-theoretic techniques are needed here, and this cannot be developed abstracting away from the underlying institution and from the specification system in use — so we defer further discussion of this to Sect. 3.2 below.

3.2 Structured Specifications

Validation techniques, no matter how powerful and useful, cannot replace in general human intuition and understanding of specifications. To support this, specifications must not be too large and too unmanageable. The idea is to keep them as readable as possible, by adapting to the realm of specification formalisms the modularity ideas of programming-in-the-large, and so by building specifications themselves in a well-structured fashion. This was perhaps first put explicitly forward in the work on CLEAR [12], and since then has been adopted as the principle behind the design of numerous specification languages. Moreover, it is a good practice to separate the issues concerned with the details of the underlying logical system from the structuring mechanisms used to build complex specifications by manipulating and putting together simpler and presumably well-understood ones. This again originates with CLEAR, and has been further put forward in [32] by insisting that *specification-building operations* should be defined in a manner as independent of the underlying institution as possible.

We will continue working here within a fixed, but otherwise quite arbitrary institution $\mathbf{I} = (\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \langle \models_\Sigma \rangle_{\Sigma \in |\mathbf{Sign}|})$. In this framework, we introduce and have a closer look at a few very simple, but flexible and quite representative specification constructs, which form the backbone of essentially all typical specification languages:

Basic specification: indicates a signature and lists axioms to capture the desirable properties.

For any signature $\Sigma \in |\mathbf{Sign}|$ and set of sentences $\Phi \subseteq \mathbf{Sen}(\Sigma)$, $\langle \Sigma, \Phi \rangle$ is a specification with $Sig[\langle \Sigma, \Phi \rangle] = \Sigma$ and $Mod[\langle \Sigma, \Phi \rangle] = Mod[\Phi]$.

Union: combines constraints imposed by various specifications.

For any specifications SP_1 and SP_2 with $Sig[SP_1] = Sig[SP_2]$, $SP_1 \cup SP_2$ is a specification with $Sig[SP_1 \cup SP_2] = Sig[SP_1]$ and $Mod[SP_1 \cup SP_2] = Mod[SP_1] \cap Mod[SP_2]$.

Translation: renames and introduces new components, following a signature morphism.

For any specification SP and signature morphism $\sigma\colon Sig[SP] \to \Sigma'$, $\sigma(SP)$ is a specification with $Sig[\sigma(SP)] = \Sigma'$ and $Mod[\sigma(SP)] = \{M' \in |\mathbf{Mod}(\Sigma')| \mid M'|_\sigma \in Mod[SP]\}$.

Hiding: hides auxiliary components, receding to the source of a signature morphism.

For any specification SP' and signature morphism $\sigma\colon \Sigma \to Sig[SP']$, $SP'|_\sigma$ is a specification with $Sig[SP'|_\sigma] = \Sigma$ and $Mod[SP'|_\sigma] = \{M'|_\sigma \mid M' \in Mod[SP']\}$.

In a way, basic specifications are directly available in the underlying institution and it is hard to imagine a specification formalism that would not include them (at least for finite sets of axioms). One interesting question here is whether the use of other specification-building operations as defined above allows one to go beyond the specification capability of basic specifications. This is indeed the case, with the essential new power added by the hiding operation: for instance, in the equational institution $\mathbf{EQ}$, classes of models of simple specifications of the form $\langle\Sigma, \emptyset\rangle|_\iota$ for some algebraic signature Σ and signature inclusion $\iota\colon \Sigma_{sub} \to \Sigma$, are not in general closed under subalgebras, and so cannot be defined by a basic specification (cf. Fact 3). The following "normal form" results give a more complete answer:

Fact 6. *Any specification built out of basic specifications using union and translation only is equivalent to a basic specification.*

Fact 7. *If the category of signatures has pushouts and the institution admits amalgamation, then any specification SP built out of basic specifications using union, translation and hiding may be equivalently transformed to its* normal form*:*

$$\mathbf{nf}(SP) = \langle\Sigma_{all}, \Phi_{all}\rangle|_{\sigma_{res}}$$

for some signature Σ_{all}, set $\Phi_{all} \subseteq \mathbf{Sen}(\Sigma_{all})$ and signature morphism $\sigma_{res}\colon Sig[SP] \to \Sigma_{all}$.

Informally, Σ_{all} is an "overall" signature obtained by combining all the "intermediate" signatures involved in SP, σ_{res} indicates how the result signature is included there, and Φ_{all} collects all the axioms used within SP.

While such normal forms provide an extremely useful technical tool to study specifications and their semantic properties, they must not be overemphasized and offered to the user too eagerly. For instance, let us have a look at the problem of deriving semantic consequences of specifications.

Recall that a sentence $\varphi \in \mathbf{Sen}(Sig[SP])$ is a *semantic consequence* of a specification SP, written $SP \models \varphi$, if φ holds in every model of SP. Any proof-theoretic counterpart of this must rely, of course, on some proof-theoretic entailment $\langle\vdash_\Sigma\rangle_{\Sigma\in|\mathbf{Sign}|}$ added to the underlying institution as in Sect. 2.4. The following rule extends this to consequences of structured specifications:

$$\frac{\mathbf{nf}(SP) = \langle\Sigma_{all}, \Phi_{all}\rangle|_{\sigma_{res}} \qquad \Phi_{all} \vdash_{\Sigma_{all}} \sigma_{res}(\varphi)}{SP \vdash \varphi}$$

This is sound and complete for the semantic consequence of specifications whenever the category of signatures has pushouts, the institution admits amalgamation (so that $\mathbf{nf}(SP) \equiv SP$ can be obtained), and $\vdash_{\Sigma_{all}}$ is sound and complete.

BUT: this is a "bad" way of proving consequences of structured specifications. The point is that no use is made of the specification structure and proofs are reduced to the proof search for consequences of a typically huge (and totally unstructured here) set Φ_{all}. A much better proof system is given by the rules in Fig. 9. Following [32] (cf. [11] for full analysis), we give a proof rule for each of the specification-building operations, allowing the user to reduce deriving consequences of the overall specification to deriving consequences of its components, via a structural rule that links the system with proof-theoretic entailment for the underlying institution (the last rule in Fig. 9). Intuitively, the resulting *compositional* proof system facilitates proof search by directing the user to the "right" components of the overall specification, thus allowing the irrelevant parts to be disregarded. This can be exploited in a proof support system for structured specifications, as perhaps first indicated in [31].

$$\frac{\varphi \in \Phi}{\langle \Sigma, \Phi \rangle \vdash \varphi} \qquad \frac{SP_1 \vdash \varphi}{SP_1 \cup SP_2 \vdash \varphi} \qquad \frac{SP_2 \vdash \varphi}{SP_1 \cup SP_2 \vdash \varphi}$$

$$\frac{SP \vdash \varphi}{\sigma(SP) \vdash \sigma(\varphi)} \qquad \frac{SP' \vdash \sigma(\varphi)}{SP'|_\sigma \vdash \varphi}$$

$$\frac{SP \vdash \varphi_1 \quad \cdots \quad SP \vdash \varphi_n \qquad \{\varphi_1, \ldots, \varphi_n\} \vdash \varphi}{SP \vdash \varphi}$$

Figure 9: Compositional proof system for consequences of structured specifications

Fact 8. *The proof system of Fig. 9 is sound if entailment $\langle \vdash_\Sigma \rangle_{\Sigma \in |\mathbf{Sign}|}$ is sound: for any structured specification SP and sentence $\varphi \in \mathbf{Sen}(Sig[SP])$, $SP \vdash \varphi \implies SP \models \varphi$.*

For institutions with pushouts of signatures admitting amalgamation and interpolation and equipped with a sound and complete entailment $\langle \vdash_\Sigma \rangle_{\Sigma \in |\mathbf{Sign}|}$, the proof system is also complete: for any structured specification SP built out of basic specifications using union, translation and hiding, and sentence $\varphi \in \mathbf{Sen}(Sig[SP])$, $SP \vdash \varphi \iff SP \models \varphi$.

The most troublesome assumption for the completeness of the compositional proof system of Fig. 9, the interpolation property, cannot be avoided here at least in some form. In general, there can be no sound and complete compositional proof system for the specifications considered. As a consequence, some mixed proof systems have to be used, which allow for the use of normal forms for structured specifications only if really necessary, maintaining the structure of specifications to guide the proof search as much as possible [28].

The proof system for consequences of structured specifications may be used also to establish that one specification is stronger than another; the corresponding compositional rules are given in Fig. 10, cf. [11]. Again, this proof system is sound and about the best it can be, with completeness following for the cases where the system of Fig. 9 is complete. The rule for hiding is a bit unusual in using a "conservative extension" requirement formulated semantically — this cannot be avoided in general, though some "proof-theoretic" approximations are possible (and the use of "definitional extensions" is most common).

3.3 Structured Specifications in CASL

We give an example of a structured specification using the notation of CASL [1, 14], extending what we have used for writing specification examples in Sect. 2.2. Further specification-building operations of CASL do not really go beyond the operations of Sect. 3.2, but provide a considerably more convenient notation and abbreviations for commonly-used specification schemes.

$$\frac{SP \vdash \varphi, \text{ for each } \varphi \in \Phi \subseteq \mathbf{Sen}(Sig[SP])}{SP \vdash \langle \Sigma, \Phi \rangle}$$

$$\frac{SP \vdash SP_1 \qquad SP \vdash SP_2}{SP \vdash SP_1 \cup SP_2} \qquad \frac{SP\big|_\sigma \vdash SP'}{SP \vdash \sigma(SP')}$$

$$\frac{SP_{ce} \vdash SP' \text{ with } \forall M \in Mod[SP]. \exists M_{ce} \in SP_{ce}. M_{ce}\big|_\sigma = M}{SP \vdash SP'\big|_\sigma}$$

Figure 10: Compositional proof system for consequences between structured specifications

- Union of specifications is written as SP_1 **and** SP_2; it allows for specifications over different signatures, by translating them implicitly to the (set-theoretic) union signature.
- Translation of a specification is written as SP **with** $s \mapsto s', f \mapsto f', \ldots$, where the "**with**-list" describes the (necessarily surjective) signature morphism.
- Union and translation may be combined to build a specification *extension*, written as SP **then** ΔSP, which puts SP into a larger context of additional signature components and further constraints introduced by ΔSP.
- Hiding is limited to subsignatures only, and written by listing either signature symbols that are left visible or those that are hidden, as SP' **reveal** $s, f, \ldots$ or SP' **hide** $s', f', \ldots$, respectively.
- Parameterized specifications are provided, where some part of a specification can be indicated as a formal parameter, to be substituted for when the parameterized specification is used by an actual argument specification, perhaps fitted using a signature morphism. (This comes with a mechanism of compound identifiers and their renaming at instantiation, which could be used to simplify the presentation below.)

We refer to [14] for further details, and hope that the above explanations are sufficient to look through the following example, motivated by the problem of searching for a "motif" in DNA sequences (hence somewhat pretentious terminology here), see [30][3]. In essence, we specify a function that given a set of sequences, finds a pattern of each length that is most similar to some subsequence of each of the sequences in the set. (We refer to Sect. 2.2 for the specification NAT and omit the standard axioms for inequalities on Nat; Fig. 5 might help to decipher specifications DNASEQ and DNASET).

spec FULLNAT = NAT
  then preds $\_ < \_, \_ \leq \_ : Nat \times Nat$
    axioms ...

spec NUCLEOTIDES = **free type** $Nucl ::= A \mid C \mid G \mid T$

spec ELEM = **sort** $Elem$

[3]Many thanks to Jerzy Tiuryn for an interesting lecture on this problem.

spec ARRAY[ELEM] = ELEM **and** NAT
  then sort $Array$
    ops $empty : Array;$
      $size : Array \to Nat;$
      $put : Nat \times Elem \times Array \to Array;$
      $get : Nat \times Array \to Elem$
    $\forall\, i, j : Nat; e : Elem; a : Array$
      - $size(empty) = 0$
      - $i < size(a) \implies size(put(i, e, a)) = size(a)$
      - $i = size(a) \implies size(put(i, e, a)) = succ(size(a))$
      - $i \leq size(a) \implies get(i, put(i, e, a)) = e$
      - $i < size(a) \wedge i \neq j \implies get(i, put(j, e, a)) = get(i, a)$

spec DNASEQ = ARRAY[NUCLEOTIDES **fit** $Elem \mapsto Nucl$] **with** $Array \mapsto DNASeq$

spec DNASET = ARRAY[DNASEQ **fit** $Elem \mapsto DNASeq$] **with** $Array \mapsto DNASet$

spec DIFF = DNASET
  then op $diff : DNASeq \times Nat \times DNASeq \times Nat \to Nat$
    $\forall\, pat, seq : DNASeq; n, m : Nat$
      - $m = size(pat) \implies diff(pat, m, seq, n) = 0$
      - $n = size(seq) \wedge m < size(pat) \implies size(pat) \leq diff(pat, m, seq, n) + m$
      - $n < size(seq) \wedge m < size(pat) \wedge get(m, pat) = get(n, seq) \implies$
        $diff(pat, m, seq, n) = diff(pat, succ(m), seq, succ(n))$
      - $n < size(seq) \wedge m < size(pat) \wedge get(m, pat) \neq get(n, seq) \implies$
        $diff(pat, m, seq, n) = succ(diff(pat, succ(m), seq, succ(n)))$

spec DISTANCES = DIFF
  then ops $set\_dist : DNASeq \times DNASet \to Nat;$
      $dist : DNASeq \times DNASeq \to Nat$
    $\forall\, set : DNASet; pat, seq : DNASeq$
      - $0 < size(set) \implies \exists i{:}Nat \bullet i < size(set) \wedge$
        $set\_dist(pat, set) = dist(pat, get(i, set))$
      - $\forall i{:}Nat \bullet i < size(set) \implies dist(pat, get(i, set)) \leq set\_dist(pat, set)$
      - $\exists i{:}Nat \bullet i \leq size(seq) \wedge dist(pat, seq) = diff(pat, 0, seq, i)$
      - $\forall i{:}Nat \bullet i < size(seq) \implies dist(pat, seq) \leq diff(pat, 0, seq, i)$

spec BESTMOTIF = DISTANCES
  then ops $best\_motif : Nat \times DNASet \to DNASeq;$
      $dist\_bound : Nat \times DNASet \to Nat$
    $\forall\, set : DNASet; n : Nat; pat : DNASeq$
      - $size(best\_motif(n, set)) = n$
      - $set\_dist(best\_motif(n, set), set) = dist\_bound(n, set)$
      - $dist\_bound(size(pat), set) \leq set\_dist(pat, set)$

spec USERBESTMOTIF = BESTMOTIF **hide** $diff, dist, set\_dist$

Given a requirements specification, produce a module that correctly implements it

The above formulation of a development task that a programmer faces suffers from the obvious lack of precision. Working again with an institution $\mathbf{I} = (\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \langle \models_\Sigma \rangle_{\Sigma \in |\mathbf{Sign}|})$ with models capturing the intended semantic features of programs to be developed, in a specification framework as presented above, this can take the following precise form:

Given a requirements specification SP, build a model $M \in |\mathbf{Mod}(Sig[SP])|$ such that $M \in Mod[SP]$

The key idea is that this should not be carried out by a single jump from a high-level user-oriented requirements specification to a final program. This simply cannot work for tasks of practical size and complexity. Instead, one should proceed step by step, adding gradually more and more detail and incorporating more and more design and implementation decisions, so that a chain of *refinement steps* is obtained, leading to a specification that is easy to implement directly:

$$SP_0 \rightsquigarrow SP_1 \rightsquigarrow \cdots \rightsquigarrow SP_n$$

For specifications SP and SP', we write $SP' \rightsquigarrow SP$ to capture that fact that SP is a *refinement* of SP'. Formally, we require that refinements preserve static interfaces (adjustments may be made using various forms of hiding) and that gradually more and more constraints are imposed, that is: $Sig[SP'] = Sig[SP]$ and $Mod[SP] \subseteq Mod[SP']$. Correctness of such a procedure is self-evident:

$$\frac{SP_0 \rightsquigarrow SP_1 \rightsquigarrow \cdots \rightsquigarrow SP_n \qquad M \in Mod[SP_n]}{M \in Mod[SP_0]}$$

The task of building up the model M and then verifying that indeed $M \in Mod[SP_0]$ is decomposed here into simpler tasks of inventing and verifying relatively minor individual refinement steps $SP_i \rightsquigarrow SP_{i+1}$. The proof system for consequences between specifications, as given in Fig. 10 (and involving that of Fig. 9) may be used for the latter purpose.

We can increase the clarity of the presentation of the development process by explicitly separating those parts of the specifications involved that become fixed, entirely determined and are not modified in further development — think of such parts as pieces of code. To accommodate them in our view of refinement steps, we introduce another component here, a *constructor*, which intuitively corresponds to a *parameterized program* [18], or a parameterized module, like a STANDARD ML *functor* [29]. Semantically such constructors are just functions that map models to models; we will classify them by the signatures of their arguments and of their results. This leads to the following definition: a specification SP is a *constructor refinement* of SP' via a constructor $\kappa\colon |\mathbf{Mod}(Sig[SP])| \to |\mathbf{Mod}(Sig[SP'])|$, written $SP' \underset{\kappa}{\rightsquigarrow} SP$, whenever $\kappa(Mod[SP]) \subseteq Mod[SP']$. The development process now takes the form of a chain of such constructor refinements; its correctness is just as evident as for simple refinements:

$$\frac{SP_0 \underset{\kappa_1}{\rightsquigarrow} SP_1 \underset{\kappa_2}{\rightsquigarrow} \cdots \underset{\kappa_n}{\rightsquigarrow} SP_n = EMPTY}{\kappa_1(\kappa_2(\ldots\kappa_n(empty)\ldots)) \in Mod[SP_0]}$$

where $EMPTY$ is a specification with a given model $empty \in Mod[EMPTY]$[4].

To give a simple example of constructor refinement, let us recall specification BESTMOTIF of Sect. 3.3. One rather obvious refinement of BESTMOTIF would be to choose between various options for some of the specified operations that are still left open there. For instance, the result of the operation diff is not fully determined for the cases where the size of the pattern is larger than that of the subsequence it is compared with, and we could replace the axiom:

$$n = size(seq) \wedge m < size(pat) \implies size(pat) \leq \mathit{diff}(pat, m, seq, n) + m$$

by a stronger one, which fully determines the operation:

$$n = size(seq) \wedge m < size(pat) \implies \mathit{diff}(pat, m, seq, n) = size(pat)$$

Perhaps more interestingly, we could choose an algorithm to define the operation of finding the closest motif for a given set of sequences in terms of a simpler operation of finding a motif that differs from a subsequence of each of the sequences in the set by a specific number of nucleotide occurrences (this is in fact closer to the motif search problem [30]). Here is a specification which describes this:

spec MOTIF = DISTANCES
  then pred $is\_motif : Nat \times Nat \times DNASet$
    op $motif : Nat \times Nat \times DNASet \to DNASeq$
    $\forall\, set : DNASet; l, d : Nat; pat : DNASeq$
      $\bullet\ is\_motif(l, d, set) \iff$
        $\exists pat{:}DNASeq \bullet size(pat) = l \wedge set\_dist(pat, set) \leq d$
      $\bullet\ is\_motif(l, d, set) \implies set\_dist(motif(l, d, set), set) \leq d$

spec USERMOTIF = MOTIF **hide** $\mathit{diff}, dist, set\_dist$

Next we define a constructor, in a notation reminiscent of STANDARD ML functors:

functor $F(M$: USERMOTIF$)$: USERBESTMOTIF =
  struct open M;
    fun $dist\_bound(l, set) =$
      let fun $leastd(d) =$ **if** $is\_motif(l, d, set)$ **then** d **else** $leastd(succ(d))$
      in $leastd(0)$ **end**;
    fun $best\_motif(l, d) = motif(l, dist\_bound(l, set), set)$
  end

Then we have:

$$\text{USERBESTMOTIF} \underset{F}{\leadsto} \text{USERMOTIF}$$

3.5 Local Constructions in Global Applications

The way constructors have been used in constructor refinement steps $SP' \underset{\kappa}{\leadsto} SP$ in Sect. 3.4 suggests that they are in a sense "global" — they are applied to a model of the entire specification SP' to build as a result a model of the entire specification SP. This is not very realistic: some ways of building such "global" constructors out of "local" constructions are necessary, using only a part of the model of the refined specification and perhaps building only a part of the result as well. One rather basic technology for this is described below.

[4] The names are meant to suggest that $EMPTY$ is the specification with the "empty" (initial) signature and no axioms, which has the unique "empty" model $empty$.

Let us consider a "local" construction $F\colon |\mathbf{Mod}(\Sigma)| \to |\mathbf{Mod}(\Sigma')|$. First, we assume that the construction neither destroys nor modifies the argument (the module provided as an argument remains available as it is); we capture this by requiring that the argument signature is included in the result signature by some signature morphism $\iota\colon \Sigma \to \Sigma'$, and that the construction is *persistent*: $F(M)\big|_\iota = M$, for all $M \in |\mathbf{Mod}(\Sigma)|$.

The idea is now to use the construction F in a *global context*, an "entire" model built so far over a signature Σ_G, via a *fitting morphism* $\gamma\colon \Sigma \to \Sigma_G$. Following the intuition of Sect. 2.3, the overall result will be a model over the pushout signature for γ and ι, and can be built assuming the amalgamation property (Sect. 2.4) by first reducing the global model to the argument signature for F, then applying F, and finally putting together the result and the original global model. We thus lift F to a "global" construction F_G; see Fig. 11.

$$\begin{tikzcd}
\mathcal{G} \arrow[rr, mapsto, "F_G"] & & F_G(\mathcal{G}) \\
\Sigma_G \arrow[rr, "\iota'"] & & \Sigma'_G \\
\Sigma \arrow[u, "\gamma"] \arrow[rr, "\iota"'] \arrow[urr, phantom, "\text{PO}"] & & \Sigma' \arrow[u, "\gamma'"'] \\
\mathcal{G}\big|_\gamma \arrow[rr, mapsto, "F"'] & & F(\mathcal{G}\big|_\gamma)
\end{tikzcd}$$

Amalgamation and persistency in use!

$F_G\colon |\mathbf{Mod}(\Sigma_G)| \to |\mathbf{Mod}(\Sigma'_G)|$ is given by:

- $F_G(\mathcal{G})\big|_{\iota'} = \mathcal{G}$ and
- $F_G(\mathcal{G})\big|_{\gamma'} = F(\mathcal{G}\big|_\gamma)$.

Figure 11: From local to global construction

To specify such constructions we provide a specification for the construction's argument and a specification for its result. We write $\mathit{Mod}[\mathit{SP} \xrightarrow{\iota} \mathit{SP}']$ for the class of all local constructions $F\colon |\mathbf{Mod}(\mathit{Sig}[\mathit{SP}])| \to |\mathbf{Mod}(\mathit{Sig}[\mathit{SP}'])|$ that are persistent along $\iota\colon \mathit{Sig}[\mathit{SP}] \to \mathit{Sig}[\mathit{SP}']$ and *strictly correct* w.r.t. parameter specification SP and result specification SP', that is, satisfy $F(M) \in \mathit{Mod}[\mathit{SP}']$ for all $M \in \mathit{Mod}[\mathit{SP}]$.

The following fact shows how correctness of local constructions lifts to the global level:

Fact 9. *Under notation as above, if* $F \in \mathit{Mod}[\mathit{SP} \xrightarrow{\iota} \mathit{SP}']$, $\mathit{Mod}[\mathit{SP}_G] \subseteq \mathit{Mod}[\gamma(\mathit{SP})]$, *and* $\mathit{Mod}[\gamma'(\mathit{SP}') \cup \iota'(\mathit{SP}_G)] \subseteq \mathit{Mod}[\mathit{SP}'_G]$ *then* $F_G(\mathit{Mod}[\mathit{SP}_G]) \subseteq \mathit{Mod}[\mathit{SP}'_G]$, *that is:* $\mathit{SP}'_G \underset{F_G}{\rightsquigarrow} \mathit{SP}_G$.

The diagram in Fig. 12 might help to visualize the essence of the above fact. It involves

$$\begin{tikzcd}
\mathit{SP}'_G \arrow[r, squiggly] & \gamma'(\mathit{SP}') \cup \iota'(\mathit{SP}_G) \arrow[r, squiggly, "F_G"'] & \mathit{SP}_G \arrow[l, bend left, "\iota'"] \\
& \mathit{SP}' \arrow[u, "\gamma'"] \arrow[ur, phantom, "\text{PO}"] & \mathit{SP} \arrow[l, "\iota"] \arrow[u, "\gamma"']
\end{tikzcd}$$

Figure 12: Correctness of global refinement

both refinement steps as well as morphisms between specifications: these are just signature morphisms required to preserve the constraints imposed by specifications. More formally, a *specification morphism* $\sigma\colon \mathit{SP} \to \mathit{SP}'$ is a signature morphism $\sigma\colon \mathit{Sig}[\mathit{SP}] \to \mathit{Sig}[\mathit{SP}']$

such that $M'\big|_\sigma \in Mod[SP]$ for all $M' \in Mod[SP']$; for $SP \equiv \langle\Sigma, \Phi\rangle$ and $SP' \equiv \langle\Sigma', \Phi'\rangle$, this is equivalent to $\Phi' \models_{\Sigma'} \sigma(\Phi)$. This defines the category $\mathbf{Spec}$ of specifications and their morphisms. Moreover, the model functor of the underlying institution obviously extends to a model functor $\mathbf{Mod}\colon \mathbf{Spec}^{op} \to \mathbf{Cat}$, with $\mathbf{Mod}(SP)$ being the full subcategory of $\mathbf{Mod}(Sig[SP])$ with objects $Mod[SP]$. Assuming that the class of specifications considered is closed (up to specification equivalence) under union and translation (which covers for instance basic specifications only, as well as structured specifications of Sect. 3.2, and CASL specifications), we have the following fact, which allows us to generalize to specifications the use of colimits to put signatures together and of the amalgamation property to combine models.

Fact 10. *If the category* $\mathbf{Sign}$ *of signatures has colimits of diagrams of a given shape then so does the category* $\mathbf{Spec}$ *of specifications. Moreover, if* $\mathbf{Mod}\colon \mathbf{Sign}^{op} \to \mathbf{Cat}$ *preserves limits of diagrams of a given shape then so does its extension* $\mathbf{Mod}\colon \mathbf{Spec}^{op} \to \mathbf{Cat}$.

4 Architectural Specifications

4.1 Architectural Specifications in Program Development

Constructor refinement steps of Sect. 3.4 capture one way of identifying a self-contained module as a separate part of the final program. The construction involved in such a step (either global or local, as in Sect. 3.5) is built and put aside; further development concerns only the realization of the remaining specification. Once this is completed, the overall result consists of two well-separated parts: the construction and the realization of the remaining specification. The latter may of course be further decomposed in a similar fashion.

The obvious and useful generalization of this is that the task to implement one specification may be decomposed into development of several independent subtasks. One hope might be to use the structure of the requirements specification for this purpose, and build a realization of the entire specification out of realizations of its parts. While this might be useful and profitable in some cases, it cannot work always (try to build a realization of $SP_1 \cup SP_2$ out of *arbitrary* realizations of SP_1 and SP_2). Moreover, even when technically this turns out possible, such a strategy must not be imposed as obligatory. Quite obviously, the goals of structuring requirements specifications and criteria for their quality are quite different from those for structuring final programs and the quality of the modular structure designed in the development process (see [17] for an early explicit discussion of this point).

Consequently, instead of aiming at useful, but restrictive development rules of the form:

$$\text{NO}\quad \frac{SP'_1 \underset{\kappa_1}{\rightsquigarrow\!\!\!>} SP_1 \quad \cdots \quad SP'_n \underset{\kappa_n}{\rightsquigarrow\!\!\!>} SP_n}{\mathbf{sbo}(SP'_1, \ldots, SP'_n) \underset{???}{\rightsquigarrow\!\!\!>} \mathbf{sbo}(SP_1, \ldots, SP_n)}\quad \text{NO}$$

for all specification-building operations **sbo**, we use a more general notion of constructor implementation, involving a branching of the development process into a number of independent development tasks:

$$SP \rightsquigarrow\!\!\!> \boxed{BR \begin{cases} SP_1 \\ \vdots \\ SP_n \end{cases}}$$

Such a branching step BR involves some "linking procedure" (semantically: n-argument constructor) $\kappa_{BR}\colon |\mathbf{Mod}(Sig[SP_1])| \times \cdots \times |\mathbf{Mod}(Sig[SP_n])| \to |\mathbf{Mod}(Sig[SP])|$, where we require that $\kappa_{BR}(M_1, \ldots, M_n) \in Mod[SP]$ for all $M_1 \in Mod[SP_1], \ldots, M_n \in Mod[SP_n]$.

Further development may now proceed to implement specifications $SP_1,\ldots,SP_n$ separately, resulting in independent, self-contained modules realizing each of them. These modules may now be used (or perhaps better: combined) by the construction κ_{BR} to build a final realization of SP. Such a branching development step thus amounts to an *organizational specification* or a modular design of the structure of the system to be developed.

CASL provides an explicit way to write such specifications down in the form of an *architectural specification* [5]:

$$\textbf{arch spec } BR = \boxed{\begin{array}{ll} \textbf{units} & U_1\colon SP_1 \\ & \ldots \\ & U_n\colon SP_n \\ \textbf{result} & \kappa_{BR}(U_1,\ldots,U_n) \end{array}}$$

For notational convenience, all the *units* to be further developed are named, and the names are used in the description of the "linking procedure" that uses them to produce the overall system. More crucially, just as in the branching development steps discussed above, each such unit comes with its specification; we require that the final construction works for arbitrary unit realizations as long as they satisfy the specifications imposed, which makes further development of each unit independent from the others.

Some of the units may be *parameterized* (or *generic*) — these are local constructions discussed in Sect. 3.5. Just as non-parameterized units, they must be given precise specifications, taking the form introduced for local construction in Sect. 3.5. A correct use of such a parameterized unit applies it to an argument that satisfies the parameter specification. In CASL, the argument may be fitted to the parameter specification via a fitting morphism, much as in instantiations of parameterized specifications (see Sect. 3.3); an appropriate signature pushout defines the signature of the result, and amalgamation is used to determine the resulting model as in Sect. 3.5 for applications of local constructions to global models.

Specifications of individual units in an architectural specification provide explicit barriers for information flow from the units to the construction they are used in, and so between the units as well. The construction must be correct and yield correct overall results for all possible unit realizations as long as they satisfy their specifications. Verification of the construction may therefore rely only on information about the units explicitly given in their specifications.

To give a simple example of an architectural specification, let us recall the structured specification USERMOTIF of Sect. 3.4 and 3.3. As discussed above, the structure of this specification, aimed at facilitating its reading and understanding, must not be misread as prescribing the structure of the software system to be developed to implement it. Indeed, we certainly do not want to force the programmer to implement all the hidden operations measuring for instance the distances between patterns and sets of sequences. Perhaps less evidently, we might notice that the parameterized specification of arrays is used for two quite different purposes: first to present sequences of nucleotides and then to capture sets of such sequences (perhaps it would be better to reflect this by using a specification of sets for the latter purpose). Quite different efficiency expectations may arise at these two levels, and distinct realizations of arrays may be called for. The architectural specification below captures one possible design of a modular system to implement USERMOTIF, admitting such a possibility. Apart from the constructs hinted at above, we use also unit definitions, translations and hiding, with the obvious meaning. Moreover, in CASL a unit may "import" another unit, which is marked using a "**given**-clause" in the unit declaration. This may be explained in terms of an appropriate declaration of an (anonymous) parameterized unit and its immediate application to the imported unit, see [5].

arch spec MotifDesign =
units FN : FullNat;
Nc : Nucleotides;
$StrA$: Elem $\to$ Array[Elem] **given** FN;
$DNA = StrA[Nc$ **fit** $Elem \mapsto Nucl]$ **with** $Array \mapsto DNASeq$;
$SetA$: Elem $\to$ Array[Elem] **given** FN;
$S = SetA[DNA$ **fit** $Elem \mapsto DNASeq]$ **with** $Array \mapsto DNASet$;
M : UserMotif **given** S
result M

The above design prescribes a rather obvious way of building up a module that stores the input data for the problem considered (culminating with the unit S) and then just indicates the essential problem to be solved: build a model M for the motif search operations extending the given unit S. Perhaps a more insightful design might be to suggest the *winnowing* algorithm, where graph-theoretic techniques are used to identify cliques[5] of a graph of "close" subsequences of the sequences in the set considered and then finding the motif (if there is one) among the cliques. A few more specifications are needed to capture this.

spec Graph[Elem] = Elem
then sort $Graph$
preds $is\_node : Elem \times Graph$;
$is\_edge : Elem \times Elem \times Graph$
$\forall\, e, e' : Elem; g : Graph$
• $is\_edge(e, e', g) \implies is\_node(e, g)$
• $is\_edge(e, e', g) \iff is\_edge(e', e, g)$

spec NatPair = FullNat **then free type** $Pair ::= \langle \_, \_ \rangle (Nat; Nat)$

spec NPGraph = Graph[NatPair **fit** $Elem \mapsto Pair$]

spec ElemSet = Array[Elem] **with** $Array \mapsto ESet$

spec ElemSetSet = Array[ElemSet **fit** $Elem \mapsto ESet$] **with** $Array \mapsto ESetSet$

spec Clique = Graph[Elem] **and** ElemSetSet
then op $cliques : Graph \to ESetSet$
axioms $\ldots$ $cliques(g)$ yields the set of all the cliques in graph g $\ldots$

spec BuildTheGraph =
{Diff **and** NPGraph
then op $build\_graph : Nat \times Nat \times DNASet \to Graph$
$\forall\, l, d : Nat; set : DNASet; i, j, i', j' : Nat$
• $is\_node(\langle i, j \rangle, build\_graph(l, d, set)) \iff$
$i < size(set) \wedge j + l < size(get(i, set))$
• $is\_edge(\langle i, j \rangle, \langle i', j' \rangle, g) \iff$
$i \neq i' \wedge$
$\exists seq\_ij{:}DNASeq \bullet size(seq\_ij) = l \wedge$
$(\forall k{:}Nat \bullet k < l \implies get(k, seq\_ij) = get(k + j, get(i, set)))$
$\wedge\ diff(seq\_ij, 0, get(i', set), j') \leq d + d$
} **hide** $diff$

[5]A clique in a graph is a set of its nodes pairwise connected by edges.

Here is the architectural specification promised above:

arch spec MOTIFCLIQUEDESIGN $=$
units FN : FULLNAT;
Nc : NUCLEOTIDES;
$StrA$: ELEM $\rightarrow$ ARRAY[ELEM] **given** FN;
$DNA = StrA[Nc$ **fit** $Elem \mapsto Nucl]$ **with** $Array \mapsto DNASeq$;
$SetA$: ELEM $\rightarrow$ ARRAY[ELEM] **given** FN;
$S = SetA[DNA$ **fit** $Elem \mapsto DNASeq]$ **with** $Array \mapsto DNASet$;
NP : NATPAIR **given** FN;
G : NPGRAPH **given** NP;
BG : BUILDTHEGRAPH **given** G, S;
$NPS = SetA[NP$ **fit** $Elem \mapsto Pair]$ **with** $Array \mapsto ESet$;
$NPSS = SetA[NPS$ **fit** $Elem \mapsto ESet]$ **with** $Array \mapsto ESetSet$;
Clq : GRAPH[ELEM] **and** ELEMSETSET $\rightarrow$ CLIQUE **given** FN;
$GC = Clq[\{G$ **and** $NPSS\}$ **fit** $Elem \mapsto Pair]$;
M : USERMOTIF **given** GC, BG
result M

This may be further refined, for instance to mark explicitly the need for removal of *spurious edges* from the graph, crucial for the calculation of the cliques to be more tractable [30].

4.2 Semantics and Verification of Architectural Specifications

Consider an institution $\mathbf{I} = (\mathbf{Sign}, \mathbf{Sen}, \mathbf{Mod}, \langle \models_\Sigma \rangle_{\Sigma \in |\mathbf{Sign}|})$. We assume that the signature category has pushouts and that the institution admits amalgamation. In such a framework we formally present a small but representative subset of architectural specifications discussed above. The fragment — or rather, its syntax — is given in Fig. 13. We now sketch its formal

Architectural specifications: $ASP ::=$ **arch spec** Dcl^* **result** T

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

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

Figure 13: A fragment of the architectural specification formalism

semantics and show how correctness of such specifications may be established. The semantics is split into the static semantics, where one checks the static well-formedness of unit terms, declarations and overall architectural specifications, and the model semantics, where the "real meaning" of the constructs involved is determined.

Static Semantics

Unit terms are statically elaborated in a *static context* $C_{st} = (P_{st}, B_{st})$, where P_{st} maps parameterized unit names to signature morphisms and B_{st} maps non-parameterized unit names to their signatures. To capture sharing between various unit components, resulting from inheritance of some unit parts via for instance parameterized unit applications, we accumulate information on non-parameterized units in a single *global signature* Σ_G, and represent non-parameterized unit signatures as morphisms into this global signature, assigning them to unit names by a map $\mathcal{B}_{st}$. This results in an *extended unit contexts* $\mathcal{C}_{st} = (P_{st}, \mathcal{B}_{st}, \Sigma_G)$; a static

$$\frac{\vdash Dcl^* \rhd \mathcal{C}_{st} \qquad \mathcal{C}_{st} \vdash T \rhd (\theta\colon \Sigma_G \to \Sigma'_G, i\colon \Sigma \to \Sigma'_G)}{\vdash \textbf{arch spec } Dcl^* \textbf{ result } T \rhd (ctx(\mathcal{C}_{st}), \Sigma)}$$

$$\frac{\mathcal{C}^{\emptyset}_{st} \vdash Dcl_1 \rhd (\mathcal{C}_{st})_1 \quad \cdots \quad (\mathcal{C}_{st})_{n-1} \vdash Dcl_n \rhd (\mathcal{C}_{st})_n}{\vdash Dcl_1 \ldots Dcl_n \rhd (\mathcal{C}_{st})_n}$$

$$\frac{\begin{array}{c} U \notin (dom(P_{st}) \cup dom(\mathcal{B}_{st})) \\ \Sigma'_G \text{ is the coproduct of } \Sigma_G \text{ and } Sig[SP] \\ \text{with injections } \theta\colon \Sigma_G \to \Sigma'_G, i\colon Sig[SP] \to \Sigma'_G \end{array}}{(P_{st}, \mathcal{B}_{st}, \Sigma_G) \vdash U\colon SP \rhd (P_{st}, (\mathcal{B}_{st};\theta) + \{U \mapsto i\}, \Sigma'_G)}$$

$$\frac{\iota : Sig[SP_1] \to Sig[SP_2] \qquad U \notin (dom(P_{st}) \cup dom(\mathcal{B}_{st}))}{(P_{st}, \mathcal{B}_{st}, \Sigma_G) \vdash U\colon SP_1 \xrightarrow{\iota} SP_2 \rhd (P_{st} + \{U \mapsto \iota\}, \mathcal{B}_{st}, \Sigma_G)}$$

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

$$\frac{\begin{array}{c} (P_{st}, \mathcal{B}_{st}, \Sigma_G) \vdash T \rhd (\theta\colon \Sigma_G \to \Sigma'_G, i\colon \Sigma_T \to \Sigma'_G) \\ P_{st}(U) = \iota\colon \Sigma \to \Sigma' \qquad \sigma\colon \Sigma \to \Sigma_T \\ (\iota'\colon \Sigma_T \to \Sigma'_T, \sigma'\colon \Sigma' \to \Sigma'_T) \text{ is the pushout of } (\sigma, \iota) \\ (\iota''\colon \Sigma'_G \to \Sigma''_G, i'\colon \Sigma'_T \to \Sigma''_G) \text{ is the pushout of } (i, \iota') \end{array}}{(P_{st}, \mathcal{B}_{st}, \Sigma_G) \vdash U[T \textbf{ fit } \sigma] \rhd (\theta;\iota''\colon \Sigma_G \to \Sigma''_G, i'\colon \Sigma'_T \to \Sigma''_G)}$$

$$\frac{\begin{array}{c} (P_{st}, \mathcal{B}_{st}, \Sigma_G) \vdash T_1 \rhd (\theta_1\colon \Sigma_G \to \Sigma^1_G, i_1\colon \Sigma_1 \to \Sigma^1_G) \\ (P_{st}, \mathcal{B}_{st}, \Sigma_G) \vdash T_2 \rhd (\theta_2\colon \Sigma_G \to \Sigma^2_G, i_2\colon \Sigma_2 \to \Sigma^2_G) \\ \Sigma = \Sigma_1 \cup \Sigma_2 \text{ with inclusions } \iota_1\colon \Sigma_1 \to \Sigma, \iota_2\colon \Sigma_2 \to \Sigma \\ (\theta'_2\colon \Sigma^1_G \to \Sigma'_G, \theta'_1\colon \Sigma^2_G \to \Sigma'_G) \text{ is the pushout of } (\theta_1, \theta_2) \\ j\colon \Sigma \to \Sigma'_G \text{ satisfies } \iota_1;j = i_1;\theta'_2 \text{ and } \iota_2;j = i_2;\theta'_1 \end{array}}{(P_{st}, \mathcal{B}_{st}, \Sigma_G) \vdash T_1 \textbf{ and } T_2 \rhd (\theta_1;\theta'_2, j)}$$

Figure 14: Static semantics rules

unit context $ctx(\mathcal{C}_{st})$ may be easily extracted from such an extended one. Given a morphism $\theta\colon \Sigma_G \to \Sigma'_G$ that extends the global signature, we write $\mathcal{B}_{st};\theta$ for the corresponding extension of $\mathcal{B}_{st}$ (mapping each $U \in dom(\mathcal{B}_{st})$ to $\mathcal{B}_{st}(U);\theta$). $\mathcal{C}^{\emptyset}_{st}$ is the "empty" extended static context (with the initial global signature). We are ready to present the *judgments* of the static semantics, one for each syntactic category of the formalism:

$\boxed{\vdash ASP \rhd (C_{st}, \Sigma)}$

Architectural specifications yield a static context and the signature of the result unit.

$\boxed{\mathcal{C}_{st} \vdash Dcl \rhd \mathcal{C}'_{st}}$

In an extended static context, unit declarations yield a new extended static context.

$\boxed{(P_{st}, \mathcal{B}_{st}, \Sigma_G) \vdash T \rhd (\theta\colon \Sigma_G \to \Sigma'_G, i\colon \Sigma \to \Sigma'_G)}$

In an extended static context, unit terms yield the unit signature embedded into a new global context signature, obtained as an indicated extension of the old one.

The semantic rules to derive these judgments are in Fig. 14, with diagrams helping to read the more complicated rules for unit application and amalgamation in Fig. 15. The use of signature union requires a bit more structure on the signature category (as in [27], for instance), but the intended meaning should be clear.

$$
\begin{tikzcd}
\Sigma_G \arrow[r, "\theta"] & \Sigma'_G \arrow[r, "\iota''"] & \Sigma''_G \\
& \Sigma_T \arrow[u, "i"] \arrow[r, "\iota'"] & \Sigma'_T \arrow[u, "i'"'] \\
& \Sigma \arrow[u, "\sigma"] \arrow[r, "\iota"'] & \Sigma' \arrow[u, "\sigma'"']
\end{tikzcd}
\qquad
\begin{tikzcd}
& \Sigma^1_G \arrow[rrdd, "\theta'_2"] & & \\
& \Sigma_1 \arrow[u, "i_1"] \arrow[rd, "\iota_1"] & & \\
\Sigma_G \arrow[ruu, "\theta_1"] \arrow[rdd, "\theta_2"'] & & \Sigma_1 \cup \Sigma_2 \arrow[r, "j"] & \Sigma'_G \\
& \Sigma_2 \arrow[d, "i_2"'] \arrow[ru, "\iota_2"'] & & \\
& \Sigma^2_G \arrow[rruu, "\theta'_1"'] & &
\end{tikzcd}
$$

Figure 15: Diagrams for unit application and amalgamation

Model Semantics

The basic concept for the model semantics of architectural specifications is that of a *unit environment* E that maps unit names to either local constructions (for parameterized units) or to individual models (for non-parameterized units). Then *unit contexts* $\mathcal{C}$ are sets of such environments, and *unit evaluators* UEv map unit environments to models. An environment E *fits* an extended static context $\mathcal{C}_{st} = (P_{st}, \mathcal{B}_{st}, \Sigma_G)$ if for $P_{st}(U) = \Sigma \xrightarrow{\iota} \Sigma'$, $E(U)$ is a local construction persistent along ι, and there exists $\mathcal{G} \in |\mathbf{Mod}(\Sigma_G)|$ such that for $U \in dom(\mathcal{B}_{st})$, $E(U) = \mathcal{G}|_{\mathcal{B}_{st}(U)}$ (then $\mathcal{G}$ *witnesses* E). We generalize this to unit contexts: $\mathcal{C}$ *fits* $\mathcal{C}_{st}$ if each $E \in \mathcal{C}$ fits $\mathcal{C}_{st}$. $\mathcal{C} \times \{U \mapsto \mathcal{V}\}$ stands for $\{E + \{U \mapsto V\} \mid E \in \mathcal{C}, V \in \mathcal{V}\}$. $\mathcal{C}^{\emptyset}$ is the set of all unit environments that fit $\mathcal{C}^{\emptyset}_{st}$.

The judgments of the model semantics take the following forms:

$\boxed{\vdash ASP \Rightarrow (\mathcal{C}, UEv)}$
Architectural specifications yield a context and an evaluator that determines the result.

$\boxed{\mathcal{C} \vdash Dcl \Rightarrow \mathcal{C}'}$
In a context, unit declarations yield a new context.

$\boxed{\mathcal{C} \vdash T \Rightarrow UEv}$
In a context, unit terms yield a unit evaluator.

The semantic rules to derive these judgments are in Fig. 16. The model semantics rules assume that the static semantics has been successful on the constructs considered. One can check that the following invariants are maintained:

$\boxed{\vdash ASP \rhd\!\!\!\rhd (C_{st}, \Sigma) \quad \vdash ASP \Rightarrow (\mathcal{C}, UEv)}$
There is an extended static context $\mathcal{C}_{st}$ such that $ctx(\mathcal{C}_{st}) = C_{st}$, $\mathcal{C}$ fits $\mathcal{C}_{st}$, $\mathcal{C} \subseteq dom(UEv)$, and for each $E \in \mathcal{C}$, $UEv(E) \in |\mathbf{Mod}(\Sigma)|$.

$\boxed{\mathcal{C}_{st} \vdash Dcl \rhd\!\!\!\rhd \mathcal{C}'_{st} \quad \mathcal{C} \vdash Dcl \Rightarrow \mathcal{C}'}$
If $\mathcal{C}$ fits $\mathcal{C}_{st}$ then $\mathcal{C}'$ fits $\mathcal{C}'_{st}$.

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

$$\frac{\mathcal{C}^\emptyset \vdash Dcl_1 \Rightarrow \mathcal{C}_1 \qquad \cdots \qquad \mathcal{C}_{n-1} \vdash Dcl_n \Rightarrow \mathcal{C}_n}{\vdash Dcl_1 \ldots Dcl_n \Rightarrow \mathcal{C}_n}$$

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

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

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

$$\frac{\begin{array}{c}\mathcal{C} \vdash T \Rightarrow UEv \\ UEv' = \{E \mapsto M \mid E \in \mathcal{C}, M\big|_{\iota'} = UEv(E), M\big|_{\sigma'} = E(U)(UEv(E)\big|_{\sigma})\}\end{array}}{\mathcal{C} \vdash U[T \textbf{ fit } \sigma] \Rightarrow UEv'}$$

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

Figure 16: Model semantics rules

$\boxed{\mathcal{C}_{st} \vdash T \rhd (\theta\colon \Sigma_G \to \Sigma'_G, i\colon \Sigma \to \Sigma'_G) \quad \mathcal{C} \vdash T \Rightarrow UEv}$

If $\mathcal{C}$ fits $\mathcal{C}_{st}$ then for each $E \in \mathcal{C}$ and $\mathcal{G}$ that witnesses E, there exists $\mathcal{G}' \in |\mathbf{Mod}(\Sigma'_G)|$ such that $\mathcal{G}'\big|_\theta = \mathcal{G}$ and $UEv(E) = \mathcal{G}'\big|_i$.

It follows in particular that the static semantics allows us to eliminate the crossed out assumptions in the model semantics rule for unit amalgamation.

Verification

The basic idea behind verification for architectural specifications is that we want to extend the static information about the units to capture their properties by an additional specification. We use the concept of a *verification context* $C_v = (P_v, SP_G)$ on an extended static context $\mathcal{C}_{st} = (P_{st}, \mathcal{B}_{st}, \Sigma_G)$, where P_v maps parameterized unit names to specification morphisms that coincide with those indicated by P_{st} (as signature morphisms), and SP_G is a *global specification* with $Sig[SP_G] = \Sigma_G$. Such a verification context therefore collects the information about individual non-parameterized units within a specification for the global signature. This is reflected in the forms of verification judgments:

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

Architectural specifications yield a specification of the result.

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

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

$\boxed{C_v \vdash T :: \Sigma \xrightarrow{i} SP'_G}$

In a verification context, unit terms yield their signature embedded into a new global specification.

The verification rules to derive these judgments are in Fig. 17. The rules assume a successful run of the static semantics and, somewhat informally, use the symbols introduced in the corresponding rules there. $C_v^{\emptyset}$ is the "empty" verification context on $\mathcal{C}_{st}^{\emptyset}$ (with the trivial global specification imposing no constraints on the models of the initial global signature).

$$\frac{\vdash Dcl^* :: C_v \qquad C_v \vdash T :: \Sigma \xrightarrow{i} SP'_G}{\vdash \textbf{arch spec } Dcl^* \textbf{ result } T :: SP'_G\big|_i}$$

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

$$\frac{}{(P_v, SP_G) \vdash U\colon SP :: (P_v, \theta(SP_G) \cup i(SP))}$$

$$\frac{}{(P_v, SP_G) \vdash U\colon SP_1 \xrightarrow{\iota} SP_2 :: (P_v + \{U \mapsto SP_1 \xrightarrow{\iota} SP_2\}, SP_G)}$$

$$\frac{\mathcal{B}_{st}(U) = \Sigma \xrightarrow{i} \Sigma_G}{(P_v, SP_G) \vdash U :: \Sigma \xrightarrow{i} SP_G}$$

$$\frac{\begin{array}{c} C_v \vdash T :: \Sigma_T \xrightarrow{i} SP'_G \\ P_v(U) = SP \xrightarrow{\iota} SP' \\ Mod[SP'_G] \subseteq Mod[i(\sigma(SP))] \end{array}}{C_v \vdash U[T \textbf{ fit } \sigma] :: \Sigma'_T \xrightarrow{i'} \iota''(SP'_G) \cup i'(\sigma'(SP'))}$$

$$\frac{C_v \vdash T_1 :: \Sigma_1 \xrightarrow{i_1} SP^1_G \qquad C_v \vdash T_2 :: \Sigma_2 \xrightarrow{i_2} SP^2_G}{C_v \vdash T_1 \textbf{ and } T_2 :: \Sigma_1 \cup \Sigma_2 \xrightarrow{j} \theta'_2(SP^1_G) \cup \theta'_1(SP^2_G)}$$

Figure 17: Verification rules

The following invariants link the results of the verification semantics with model semantics for architectural specifications, justifying the soundness of verification:

$\boxed{\vdash ASP \rhd\!\!\!\rhd (C_{st}, \Sigma) \quad \vdash ASP \Rightarrow (\mathcal{C}, UEv) \quad \vdash ASP :: SP}$

$Sig[SP] = \Sigma$ and for each $E \in \mathcal{C}$, $UEv(E) \in Mod[SP]$.

$\boxed{\mathcal{C}_{st} \vdash Dcl \rhd\!\!\!\rhd \mathcal{C}'_{st} \quad \mathcal{C} \vdash Dcl \Rightarrow \mathcal{C}' \quad (P_v, SP_G) \vdash Dcl :: (P'_v, SP'_G)}$

If $\mathcal{C}$ fits $\mathcal{C}_{st}$, (P_v, SP_G) is on $\mathcal{C}_{st}$ and every $E \in \mathcal{C}$ is witnessed by some $\mathcal{G} \in Mod[SP_G]$ then $\mathcal{C}'$ fits $\mathcal{C}'_{st}$, (P'_v, SP'_G) is on $\mathcal{C}'_{st}$ and every $E' \in \mathcal{C}'$ is witnessed by some $\mathcal{G}' \in Mod[SP'_G]$.

$\boxed{\mathcal{C}_{st} \vdash T \rhd\!\!\!\rhd (\theta\colon \Sigma_G \to \Sigma'_G, i) \quad \mathcal{C} \vdash T \Rightarrow UEv \quad (P_v, SP_G) \vdash T :: \Sigma \xrightarrow{i'} SP'_G}$

If $\mathcal{C}$ fits $\mathcal{C}_{st}$, (P_v, SP_G) is on $\mathcal{C}_{st}$ and every $E \in \mathcal{C}$ is witnessed by some $\mathcal{G} \in Mod[SP_G]$ then $Sig[SP'_G] = \Sigma'_G$, i' and i coincide as signature morphisms, and every $E \in \mathcal{C}$ is witnessed in $\mathcal{C}_{st};\theta$ by some $\mathcal{G}' \in Mod[SP'_G]$ with $UEv(E) = \mathcal{G}'\big|_i$.

5 Observational Interpretation of Specifications

5.1 Observational Preliminaries

One tricky issue in formal program specification is that even when some axioms are written down, they are not necessarily meant to literally hold in all the acceptable realizations of the specification. This is often exemplified using the standard specification of stacks, where the equation $\forall e{:}Elem, s{:}Stack.pop(push(e,s)) = s$ is not really expected in all stack realizations (e.g., it fails in the usual array-with-pointer realization of stacks, cf. Fig. 18). The

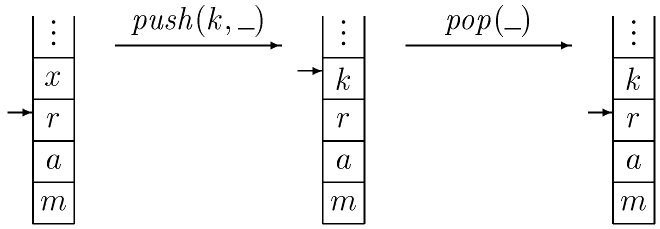

Figure 18: Array-with-pointer realization of stacks: $marx \neq mark$

point here is that even though the actual axiom does not hold in all acceptable realizations, it concerns equality between data that cannot be directly observed by the user. However, its *observable consequences*, concerning *observable results* of computations, hold in the models considered and this is sufficient to make the models acceptable. What we really care about is an externally observable *behaviour* of models and its dual side, internal *indistinguishability* of model elements (rather than their identity). The concepts that capture this more formally are discussed below.

To keep things simple, let us switch back to the standard algebraic framework, working with the usual algebraic signatures additionally assumed to contain sort $bool$ with constants $true$ and $false$ interpreted in the standard way in all the algebras considered. Moreover, given a signature $\Sigma = (S, \Omega)$, we fix a set $OBS \subseteq S$ of *observable sorts*, with $bool \in OBS$.

The intuition we want to capture is that algebras $A, B \in Alg(\Sigma)$ with $|A|_{OBS} = |B|_{OBS}$ are observationally equivalent if they exhibit the same behaviour observable via OBS: for each *observable term* $t \in |T_\Sigma(X_{OBS})|_{OBS}$ (of an observable sort in OBS, with variables X_{OBS} of observable sorts only) and valuation $v\colon X_{OBS} \to |A|_{OBS}(= |B|_{OBS}), t_A[v] = t_B[v]$.

The following notion turns out to be the key: a *correspondence* between $A, B \in Alg(\Sigma)$, $\rho\colon A \bowtie B$, is a relation $\rho \subseteq |A| \times |B|$ that is *closed under the operations*. A correspondence $\rho\colon A \bowtie B$ is *observational* if it is bijective on sorts in OBS. We define $A, B \in Alg(\Sigma)$ to be *observationally equivalent* w.r.t. OBS, written $A \equiv_{OBS} B$, if there is an observational correspondence $\rho\colon A \bowtie B$ between them.

Equipped with these definitions, we can relax the interpretation of specifications, asking for models up to observational equivalence only, i.e., for a specification SP with $Sig[SP] = \Sigma = (S, \Omega)$ and $OBS \subseteq S$ we define its class of *observational models*:

$$Abs_{OBS}[SP] = \{A \in Alg(\Sigma) \mid A \equiv_{OBS} B, \text{ for some } B \in Mod[SP]\}.$$

For global specifications, we simply choose $OBS = \{bool\}$, omitting the index OBS in all such cases: $Abs[SP_G] = \{\mathcal{G} \in Alg(\Sigma_G) \mid \mathcal{G} \equiv \mathcal{H}, \text{ for some } \mathcal{H} \in Mod[SP_G]\}$.

Such an observational interpretation of specifications is based on the notion of observational equivalence of algebras, and hence in a sense is external to the programs we model. An alternative is to look inside each individual program, and study internal indistinguishability between elements of the corresponding algebra.

Given $A \in Alg(\Sigma)$, let $\langle A \rangle_{OBS}$ be the subalgebra of A generated by the observable carriers. Then *observational indistinguishability* on A is the largest congruence $\approx_{OBS}$ on $\langle A \rangle_{OBS}$ that is the identity on observable carriers. This links with observational equivalence via the notion of an *observational quotient*: $A/{\approx_{OBS}}$ is the quotient of $\langle A \rangle_{OBS}$ by $\approx_{OBS}$. Algebras are observationally equivalent if their observational quotients coincide (up to isomorphism).

Fact 11. *For any algebras $A, B \in Alg(\Sigma)$ and set of observable sorts OBS, the following* factorization property *holds: $A \equiv_{OBS} B \iff A/{\approx_{OBS}} \cong B/{\approx_{OBS}}$.*

The following fact gives a more explicit characterization of the internal indistinguishability, relying on the notion of a context (a term with a special "hole" variable for which the values considered may be substituted):

Fact 12. *For $a, b \in |\langle A \rangle_{OBS}|$, $a \approx_{OBS} b$ if and only if for all* observable contexts *$t \in |T_\Sigma(X_{OBS} \cup \{\Box\})|_{OBS}$ and valuations $v \colon X_{OBS} \to |A|_{OBS}$, $t_A[v[\Box \mapsto a]] = t_A[v[\Box \mapsto b]]$.*

The observable indistinguishability may be used to redefine the interpretation of equality in the algebras considered. Assuming some fixed set of observable sorts OBS, we may redefine the satisfaction relation for equations in algebras, re-interpreting equality as indistinguishability: $A \models_{\approx_{OBS}} \forall X.t = t'$ if for all $v \colon X \to |A|$, $t_A[v] \approx_{OBS} t'_A[v]$. This naturally generalizes to a new interpretation of basic specifications:

$$Mod_{\approx_{OBS}}[\langle \Sigma, \Phi \rangle] = \{A \in Alg(\Sigma) \mid A \models_{\approx_{OBS}} \varphi, \text{ for all } \varphi \in \Phi\}$$

and may be lifted further to all structured specifications (leaving the inductive definition of their semantics unchanged) — although quite soon the intuition so clear for the basic equational specifications disappears (under this interpretation, try for instance to spell out the semantics of negated equations, or of a translation of a specification along an arbitrary signature morphism). Where the intuition is clear, it coincides with the observational interpretation based on external algebra equivalence (see [4] for a more complete discussion, and [8] for an institution-independent formulation):

Fact 13. *For any equational basic specification $\langle \Sigma, \Phi \rangle$, $Abs_{OBS}[\langle \Sigma, \Phi \rangle] = Mod_{\approx_{OBS}}[\langle \Sigma, \Phi \rangle]$.*

5.2 Observational Refinements

The observational interpretation of specifications introduced above should, of course, be taken into account when looking at the process of software development. This can be achieved by relaxing appropriately requirements in the constructor implementation steps. A specification SP is an *observational refinement* of SP' via a constructor $\kappa \colon |\mathbf{Mod}(Sig[SP])| \to |\mathbf{Mod}(Sig[SP'])|$, written $SP' \overset{\equiv}{\underset{\kappa}{\rightsquigarrow}} SP$, whenever $\kappa(Mod[SP]) \subseteq Abs[SP']$.

Have we made a mistake? Under the above definition, observational refinements do not compose in general:

$$\text{NO}\ \frac{SP'' \overset{\equiv}{\underset{\kappa'}{\rightsquigarrow}} SP' \qquad SP' \overset{\equiv}{\underset{\kappa}{\rightsquigarrow}} SP}{SP'' \overset{\equiv}{\underset{\kappa;\kappa'}{\rightsquigarrow}} SP}\ \text{NO}$$

Indeed, we attempt here to have our cake and eat it too: when using a model of a specification, we rely on the specification literally (considering $Mod[SP]$); when building a model for a specification, we admit doing so up to observational equivalence only (allowing the construction to yield results in $Abs[SP']$). Perhaps surprisingly, this can be made to work under an additional assumption [38]. A constructor $\kappa \colon Alg(\Sigma) \to Alg(\Sigma')$ is *stable* if it preserves observational equivalence of algebras: for all $A, B \in Alg(\Sigma)$, $A \equiv B \implies \kappa(A) \equiv \kappa(B)$. If we now assume all the constructors involved to be stable, then observational refinements compose:

$$\frac{SP'' \overset{\equiv}{\underset{\kappa'}{\rightsquigarrow}} SP' \qquad SP' \overset{\equiv}{\underset{\kappa}{\rightsquigarrow}} SP}{SP'' \overset{\equiv}{\underset{\kappa;\kappa'}{\rightsquigarrow}} SP}$$

Consequently, the development process with the notion of refinement relaxed as above yields correct results:

$$\frac{SP_0 \overset{\equiv}{\underset{\kappa_1}{\rightsquigarrow}} SP_1 \overset{\equiv}{\underset{\kappa_2}{\rightsquigarrow}} \cdots \overset{\equiv}{\underset{\kappa_n}{\rightsquigarrow}} SP_n = EMPTY}{\kappa_1(\kappa_2(\ldots \kappa_n(empty)\ldots)) \in Abs[SP_0]}$$

Unfortunately, stability is not a well-behaving property: many standard ways of building constructions may lead from stable to non-stable constructions. This applies, for instance, to lifting a local construction to a global context over a signature pushout, as discussed in Sect. 3.5 (cf. Fig. 11). Therefore, we replace stability by its stronger version, so called *local stability*, which requires constructions to preserve not only observational equivalence but also its potential witnesses, correspondences. $F\colon Alg(\Sigma) \to Alg(\Sigma')$ is *locally stable* if for every correspondence $\rho\colon A \bowtie B$ on Σ-algebras, there exists $\rho'\colon F(A) \bowtie F(B)$ that extends ρ, i.e., $\rho'\big|_\iota = \rho$. Now, it is easy to check that local stability allows the construction to be safely used in an arbitrary context: lifting a locally stable construction over a pushout, as in Fig. 11, yields a locally stable, and hence stable, construction.

The notion of correctness of local constructions requires similar adjustments. We write $Mod_{lc}[SP \xrightarrow{\iota} SP']$ for the class of all local constructions $F\colon Alg(Sig[SP]) \to Alg(Sig[SP'])$ that are persistent along $\iota\colon Sig[SP] \to Sig[SP']$, locally stable, and *observationally correct* w.r.t. parameter specification SP and result specification SP', that is, such that for each $A \in Mod[SP]$ there exists a correspondence $\rho'\colon F(A) \bowtie A'$ with $A' \in Mod[SP']$ and $\rho'\big|_\iota = id_A$. Fact 9 has its observational version [6]:

Fact 14. *Under notation as above (see Fig. 11), if* $F \in Mod_{lc}[SP \xrightarrow{\iota} SP']$, $Mod[SP_G] \subseteq Abs[SP_G \cup \gamma(SP)]$, *and* $Mod[\gamma'(SP') \cup \iota'(SP_G)] \subseteq Abs[SP'_G]$ *then* $F_G(Mod[SP_G]) \subseteq Abs[SP'_G]$, *that is:* $SP'_G \overset{\equiv}{\underset{F_G}{\rightsquigarrow}} SP_G$.

Local stability is not just a technical requirement imposed to obtain the expected technical results. Under closer scrutiny, it turns out that locally stable constructions are those that respect *modularity* in the software development process, never attempting to use their arguments in a way that would break visibility rules for instance by checking the details of the given realization of argument data types. Very informally (this can be made formal, see [6] for some discussion) conditional definitions yield a locally stable construction if all the branching conditions are *observable* (in the present framework: are equalities between terms of the sort $bool$). So, local stability is not a property one would be expected to check for each construction separately; it should rather follow from the design of the programming language used for coding up constructions. Once this is checked for such a language, there is no need to bother checking stability during the development process. In a word, stability is not a constraint imposed on the developer, but a principle the language designer should follow:

Local stability: *good language design directive*

5.3 Observational Interpretation of Architectural Specifications

Given the observational interpretation of specifications and the local stability requirement imposed on local constructions, architectural specifications discussed in Sect. 4 require some adjustments as well. Of course, the overall methodological ideas and justification for their use still apply. Perhaps surprisingly, not much change is required to the semantics and verification either. We illustrate this for the simple architectural specifications discussed in detail in Sect. 4.2; recall the syntax of the formalism given in Fig. 13.

Static Semantics

The static semantics remains just the same as in Sect. 4.2: the observational interpretation does not modify the static properties (signatures, signature morphisms) of constructs in architectural specifications at all.

Observational Model Semantics

For the observational version of the model semantics we use judgments of the same form as in Sect. 4.2, with quite the same intuitive meaning, just written in a slightly modified way:

$$\boxed{\begin{array}{l} \vdash ASP \overset{\equiv}{\Longrightarrow} (\mathcal{C}, UEv) \\ \mathcal{C} \vdash Dcl \overset{\equiv}{\Longrightarrow} \mathcal{C}' \\ \mathcal{C} \vdash T \overset{\equiv}{\Longrightarrow} UEv \end{array}}$$

Among the semantic rules, we clearly need new rules for unit declarations, which under observational interpretation of the specifications involved take the following form:

$$\frac{}{\mathcal{C} \vdash U\colon SP \overset{\equiv}{\Longrightarrow} \mathcal{C} \times \{U \mapsto Abs[SP]\}}$$

$$\frac{}{\mathcal{C} \vdash U\colon SP_1 \xrightarrow{\iota} SP_2 \overset{\equiv}{\Longrightarrow} \mathcal{C} \times \{U \mapsto Mod_{lc}[SP_1 \xrightarrow{\iota} SP_2]\}}$$

All the other rules remain the same: they either propagate information through the composition of declarations, or they capture evaluation of unit terms, which is not influenced by the observational interpretation at all.

However, we need to strengthen semantic invariants, to make evident that the stability principle is not violated. To formulate this precisely, we extend the notion of observational equivalence to local constructions and unit environments. Two local constructions $F_1, F_2\colon Alg(\Sigma) \to Alg(\Sigma')$ are observationally equivalent, written $F_1 \equiv F_2$, if for each $A \in Alg(\Sigma)$ there exists $\rho'\colon F_1(A) \bowtie F_2(A)$ with $\rho'\big|_\iota = id_A$. Two environments are observationally equivalent, written $E_1 \equiv E_2$, if E_1 and E_2 have common domains and $E_1(U) \equiv E_2(U)$ for each U in the domain. Now, the semantic invariants in Sect. 4.2 are expanded with the following:

$$\boxed{\vdash ASP \rhd\!\!\!\rhd (C_{st}, \Sigma) \quad \vdash ASP \Rightarrow (\mathcal{C}, UEv)}$$

UEv is *stable*, i.e., $UEv(E_1) \equiv UEv(E_2)$ for $E_1 \equiv E_2 \in \mathcal{C}$.

$$\boxed{\mathcal{C}_{st} \vdash T \rhd\!\!\!\rhd (\theta\colon \Sigma_G \to \Sigma'_G, i\colon \Sigma \to \Sigma'_G) \quad \mathcal{C} \vdash T \Rightarrow UEv}$$

UEv is *stable*, i.e., $UEv(E_1) \equiv UEv(E_2)$ for $E_1 \equiv E_2 \in \mathcal{C}$.

Warning: unit amalgamation is *not* stable in general. So, a direct inductive proof of the above stability invariants does not go through. One way out (see [6]) is to enhance the semantics to take explicit account of (stable) constructions at the global level; a similar idea has to be used in the proofs of verification invariants below.

Observational Verification

We keep essentially the same form of verification judgments as in Sect. 4.2, just slightly modifying the notation:

$$\boxed{\begin{array}{l} \vdash ASP :\equiv: SP \\ C_v \vdash Dcl :\equiv: C'_v \\ C_v \vdash T :\equiv: \Sigma \xrightarrow{i} SP'_G \end{array}}$$

Most of the verification rules for the simple formalism given in Sect. 4.2 just carried around information from unit declarations. No wonder, these do not change under observational interpretation at all. The only exception is the rule for the application of a paramaterized unit to an argument, where a premise asserts that the argument always satisfies the parameter specification. This has to be relaxed, admitting observational interpretation of specifications, and the rule now takes the following form (cf. Fact 14 for the crucial condition):

$$\frac{\begin{array}{c} C_v \vdash T :\equiv: \Sigma_T \xrightarrow{i} SP'_G \\ P_v(U) = SP \xrightarrow{\iota} SP' \\ Mod[SP'_G] \subseteq Abs[SP'_G \cup i(\sigma(SP))] \end{array}}{C_v \vdash U[T \textbf{ fit } \sigma] :\equiv: \Sigma'_T \xrightarrow{i'} \iota''(SP'_G) \cup i'(\sigma'(SP'))}$$

All other rules remain the same.

Observational verification invariants justify that such a verification calculus is sound for the underlying model semantics:

$$\boxed{\vdash ASP \rhd\!\!\!\rhd (C_{st}, \Sigma) \quad \vdash ASP \Rightarrow (\mathcal{C}, UEv) \quad \vdash ASP :\equiv: SP}$$

$Sig[SP] = \Sigma$ and for each $E \in \mathcal{C}$, $UEv(E) \in Abs[SP]$.

$$\boxed{\mathcal{C}_{st} \vdash Dcl \rhd\!\!\!\rhd \mathcal{C}'_{st} \quad \mathcal{C} \vdash Dcl \Rightarrow \mathcal{C}' \quad (P_v, SP_G) \vdash Dcl :\equiv: (P'_v, SP'_G)}$$

If $\mathcal{C}$ fits $\mathcal{C}_{st}$, (P_v, SP_G) is on $\mathcal{C}_{st}$ and every $E \in \mathcal{C}$ is witnessed by some $\mathcal{G} \in Abs[SP_G]$ then $\mathcal{C}'$ fits $\mathcal{C}'_{st}$, (P'_v, SP'_G) is on $\mathcal{C}'_{st}$ and every $E' \in \mathcal{C}'$ is witnessed by some $\mathcal{G}' \in Abs[SP'_G]$.

$$\boxed{\mathcal{C}_{st} \vdash T \rhd\!\!\!\rhd (\theta, i) \quad \mathcal{C} \vdash T \Rightarrow UEv \quad (P_v, SP_G) \vdash T :\equiv: \Sigma \xrightarrow{i'} SP'_G}$$

If $\mathcal{C}$ fits $\mathcal{C}_{st}$, (P_v, SP_G) is on $\mathcal{C}_{st}$ and every $E \in \mathcal{C}$ is witnessed by some $\mathcal{G} \in Abs[SP_G]$ then $Sig[SP'_G] = \Sigma'_G$, i' and i coincide as signature morphisms, and every $E \in \mathcal{C}$ is witnessed by some $\mathcal{G}' \in Abs[SP'_G]$ (in $\mathcal{C}_{st}$;θ) with $UEv(E) = \mathcal{G}'\big|_i$.

6 Final Remarks

We have presented a rather formal view of abstract specification theory, intended to provide a mathematically sound basis for specification and systematic development of correct, well-structured programs. The guideline we used throughout our presentation is the quest not only for mathematical rigor and precision, but also for sufficient abstraction and generality. We believe that this is the way to capture the essence of the issues considered and thus to make the insights and techniques offered reusable in the context of many specific formalisms and notations that might be chosen for writing down specifications and coding programs.

One aspect of this is the stress on the independence of this work from the particulars of the underlying logical system, formalized as an institution. We have only touched here on what the theory of institutions offers. The interest there ranges from a highly abstract version of "abstract model theory" to the exciting issues of moving between logical systems, for instance to compare their expressive power, combine various logical systems to build more complex ones in a well-structured manner, borrow proof techniques from one logical system to another, or build heterogeneous specifications spanning a variety of logical systems used conveniently to capture various aspects of the same program in various parts of a specification or at various phases of program development ([40, 41] offer but a starting point for such topics). What we have illustrated here is that the notion of an institution may well be used to separate the institution-dependent "kernel" of a specification formalism from its institution-independent parts, concerning for instance the structuring mechanisms for both specifications and programs.

Let us re-iterate one point we have tried to make: specification structure is an important conceptual weapon to facilitate specification construction, understanding and use. The activities centered on specifications should use it as a basic tool for mastering specification complexity. Even when this principle cannot be blindly adhered to, the structure should not be forgotten, and used as much as possible, with the non-structured way of handling specifications kept as local as possible. Very much the same applies to structuring programs, their development and verification.

The theory presented in this paper might look very abstract and detached from any practical use. In a way, this is true. However, it also provides the necessary mathematical underpinnings for more practical development of specific formalisms and tools. In particular, work on CASL is very much based on these ideas, and attempts to make them usable in practice. It offers a convenient notation, rich vocabulary of useful specification schemes, a collection of examples and more serious case studies, as well as support tools that are necessary to carry out specification and development tasks of a practical complexity — see [1] and CoFI pages at `http://www.brics.dk/Projects/CoFI/`.

References

[1] E. Astesiano, M. Bidoit, H. Kirchner, G. Krieg-Brückner, P. Mosses, D. Sannella, A. Tarlecki. CASL: The Common Algebraic Specification Language. *Theoretical Computer Science* 286(2002), 153-196.

[2] J. Barwise. Axioms for abstract model theory. *Annals of Mathematical Logic* 7(1974), 221–265.

[3] J. Barwise, S. Feferman, eds. *Model-Theoretic Logics*. Springer-Verlag, 1985.

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

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

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

[7] M. Bidoit, D. Sannella, A. Tarlecki. Toward component-oriented formal software development: an algebraic approach. *Radical Innovations of Software and Systems Engineering in the Future, Proc. Monterey Workshop*, Venice, 2002.

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

[9] G. Birkhoff, J. Lipson. Heterogeneous algebras. *Journal of Combinatorial Theory* 8(1970), 115-133.

[10] T. Borzyszkowski. Generalized interpolation in CASL. *Information Processing Letters* 76(2000), 19–24.

[11] T. Borzyszkowski. Logical systems for structured specifications. *Theoretical Computer Science*, 286(2002), 197-245.

[12] R. Burstall, J. Goguen. The semantics of CLEAR, a specification language. Proc. Copenhagen Winter School on Abstract Software Specification, D. Bjørner, ed., Springer LNCS 86, 292–332, 1980.

[13] C. Chang, H. Keisler. *Model Theory*. Studies in Logic and the Foundations of Mathematics, vol. 73, North-Holland, 1973.

[14] CoFI Language Design Task Group. CASL — The Common Algebraic Specification Language — Summary. See `http://www.brics.dk/Projects/CoFI/Documents/CASL/Summary/`.

[15] A. Davis. *Software Requirements: Analysis and Specification*. Prentice Hall, 1990.

[16] H. Ehrig, B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*. EATCS Monographs on Theoretical Computer Science, vol. 6, Springer-Verlag, 1985.

[17] J. Fitzgerald, C. Jones. Modularizing the formal description of a database system. *Proc. VDM'90 Conference*, Kiel, Springer LNCS 428, 198–210, 1990.

[18] J. Goguen. Parameterized programming. *IEEE Trans. on Software Engineering* SE-10(1984), 528–543.

[19] J. Goguen, R. Burstall. Institutions: abstract model theory for specification and programming. *Journal of the Assoc. for Computing Machinery* 39(1992), 95–146.

[20] J. Goguen, J. Meseguer. Completeness of many-sorted equational logic. *Houston Journal of Mathematics* 11(1985), 307–334.

[21] J. Goguen, J. Thatcher, E. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. In: *Current Trends in Programming Methodology, IV*, R. Yeh, ed., 80–149, Prentice-Hall, 1978.

[22] M. Gordon, T. Melham, eds. *An Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[23] G. Grätzer. *Universal Algebra*. 2nd edition, Springer-Verlag, 1979.

[24] S. MacLane. *Categories for the Working Mathematician*. Springer-Verlag, 1971.

[25] D. MacQueen, D. Sannella. Completeness of proof systems for equational specifications. *IEEE Trans. on Software Engineering* SE-11(1985), 454-461.

[26] J. Meseguer. General logics. In: *Logic Colloquium '87*, H.-D. Ebbinghaus, J. Fernández-Prida, M. Garrido, D. Lascar, eds., 275–329, North-Holland, 1989.

[27] T. Mossakowski. Specifications in an arbitrary institution with symbols. *Recent Developments in Algebraic Development Techniques, 14th Intl. Workshop on Algebraic Development Techniques* WADT'99, Springer LNCS 1827, 252–270, 2000.

[28] T. Mossakowski, S. Autexier, D. Hutter. Extending development graphs with hiding. *Proc. 2nd Conf. on Fundamental Aspects of Software Engineering* FASE'01, held within ETAPS'2001, Springer LNCS 2029, 269-283, 2001.

[29] L. Paulson. *ML for the Working Programmer*. Cambridge Univ. Press (1991).

[30] P. Pevzner, Sing-Hoi Sze. Combinatorial Approaches to Finding Subtle Signals in DNA sequences. *Proc. 8th Intl. Conf. Intelligent Systems for Molecular Biology* ISMB'00, August 19-23, 2000, La Jolla/San Diego, AAAI 2000.

[31] D. Sannella, R. Burstall. Structured theories in LCF. *Proc. 8th Colloq. on Algebra and Trees in Programming* CAAP'93, L'Aquila, Italy, 377–391, Springer LNCS 159, 1983.

[32] D. Sannella, A. Tarlecki. Specifications in an arbitrary institution. *Information and Computation* 76(1988), 165–210.

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

[34] D. Sannella, A. Tarlecki. Algebraic preliminaries. In: *Algebraic Foundations of System Specification*, E. Astesiano, H.-J. Kreowski, B. Krieg-Brückner, eds., 13–30, Springer Verlag 1999.

[35] D. Sannella, A. Tarlecki. Category theory. In: D. Sannella, A. Tarlecki. *Foundations of Algebraic Specifications and Formal Program Development*, monograph under preparation. See `http://www.mimuw.edu.pl/~tarlecki/marktoberdorf/`.

[36] D. Sannella, A. Tarlecki. Working with an arbitrary logical system. In: D. Sannella, A. Tarlecki. *Foundations of Algebraic Specifications and Formal Program Development*, monograph under preparation. See `http://www.mimuw.edu.pl/~tarlecki/marktoberdorf/`.

[37] D. Sannella, M. Wirsing. Specification languages. In: *Algebraic Foundations of System Specification*, E. Astesiano, H.-J. Kreowski, B. Krieg-Brückner, eds., 243–272, Springer Verlag 1999.

[38] 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.

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

[40] A. Tarlecki. Moving between logical systems. *Recent Trends in Data Type Specification. Selected Papers. 11th Workshop on Specification of Abstract Data Types* ADT'95, Springer LNCS 1130, 478–502, 1996.

[41] A. Tarlecki. Toward heterogeneous specifications. In: *Frontiers of Combining Systems 2*, *Proc. Intl. Conference Frontiers of Combining Systems* FroCoS'98, 337–360, Research Studies Press 2000.

[42] A. Tarlecki. Institutions: an abstract framework for formal specifications. In: *Algebraic Foundations of System Specification*, E. Astesiano, H.-J. Kreowski, B. Krieg-Brückner, eds., 105–130, Springer-Verlag 1999.

[43] M. Wirsing. Algebraic specification. In: *Handbook of Theoretical Computer Science, Vol. B*, J. van Leeuwen, ed., 675–788, Elsevier Science Publishers B.V. (North Holland) 1990.