KES.U.94.5

Kestrel Institute

SPECWARE:™
Formal Support for Composing Software

by

Yellamraju V. Srinivas and Richard Jüllig

December 1994
Updated May 1995

To appear in the Proceedings of the
Conference on Mathematics of Program Construction,
Kloster Irsee, Germany, July 1995.

KESTREL INSTITUTE • 3260 HILLVIEW AVENUE • PALO ALTO, CA 94304 • (415) 493-6871

SPECWARE:™
Formal Support for Composing Software

Yellamraju V. Srinivas and Richard Jüllig
Kestrel Institute
3260 Hillview Avenue
Palo Alto, CA 94304, USA
Email: {srinivas,jullig}@kestrel.edu
Tel: (415) 493-6871, Fax: (415) 424-1807

May 15, 1995

Abstract. SPECWARE supports the systematic construction of formal specifications and their stepwise refinement into programs. The fundamental operations in SPECWARE are that of composing specifications (via colimits), the corresponding refinement by composing refinements (via sheaves), and the generation of programs by composing code modules (via colimits). The concept of diagram refinement is introduced as a practical realization of composing refinements via sheaves. Sequential and parallel composition of refinements satisfy a distributive law which is a generalization of similar compatibility laws in the literature. SPECWARE is based on a rich categorical framework with a small set of orthogonal concepts. We believe that this formal basis will enable the scaling to system-level software construction.

Table of Contents

1 Introduction

SPECWARE™ supports the systematic construction of executable programs from axiomatic specifications via stepwise refinement. The immediate motivation for the the development of SPECWARE is the desire to integrate on a common conceptual basis the capabilities of several earlier systems developed at Kestrel Institute [Jüllig 93], including KIDS [Smith 90] and DTRE [Blaine and Goldberg 91].

1.1 Reasoning about the Structure of Specifications, Refinements, and Code

The most important new aspect of the framework developed is the ability to represent explicitly the structure of specifications, refinements, and program modules. We believe that the explicit representation and manipulation of structure is crucial to scaling program construction techniques to system development.

The basis of SPECWARE is a category of axiomatic specifications and specification morphisms. Specification structure is expressed via specification diagrams, directed multi-graphs whose nodes are labeled with specifications and arcs with specification morphisms. Specification diagrams are useful both for composing specification from pieces and for inducing on a given specification a structure suitable for the design task at hand.

In SPECWARE the design process proceeds by stepwise refinement of an initial specification into executable code. The unit of refinement is an interpretation, a theorem-preserving translation of the vocabulary of a source specification into the terms of a target specification. Each interpretation reduces the problem of finding a realization for the source specification to finding a realization for the target specification. The overall result of the design process is to refine an initial specification into a program module.

Of course, it is desirable to structure the overall refinement. Progression through multiple stages requires sequential composability of refinements. Similarly, parallel composition lets us exploit the structure of specifications by putting refinements together from refinements between sub-specifications of the source and target specifications. It is for this purpose that we introduce the notion of diagram refinements in this paper: just as specification diagrams impose a component structure on specifications, so do diagram refinements make explicit the component structure of a specification refinement.

Specification refinement exploits specification structure; code generation, in turn, exploits the refinement structure. Given translations to code for the specifications that serve as the final refinement targets, SPECWARE generates a system of modules by induction on the refinement structure. Layered module construction mirrors sequential composition of refinements, and the "gluing together" of modules into larger modules reflects the (parallel) composition of specifications and refinements from components.

Our work combines ideas and notions from the fields of algebraic specifications, category theory, and sheaf theory. We believe that the use of such "heavy" formal machinery is well-justified. For instance, category theory seems ideally suited for describing the manipulation of richly detailed structures at various levels of granularity. Similarly, the sheaf-theoretic notion

of compatible families seems fundamental to and pervasive in putting systems together from interdependent components.

The ideas and concepts presented in this paper have been implemented in the SPECWARE 1.0 system, which continues to be developed. It is interesting to note that the implementation efforts seem to fare the better the more closely the implementation reflects the underlying theoretical concepts. Conversely, experimentation with the SPECWARE system has had a significant impact on the theory of diagram refinement presented here.

1.2 Outline

We briefly present our specification language in Sect. 2 and in Appendix A. The focus of this paper is the sequential and parallel composition of refinements, as described in Sect. 4. Sect. 5 discusses how sufficiently refined specifications can be translated to programs. Sect. 6 describes related work. Finally, we offer some conclusions and an outlook on future work.

2 Putting Specifications Together

The primary component of the SPECWARE workspace is the category of specifications and specification morphisms. Diagrams in this category describe system structure. Specifications can be put together via colimits to obtain more complex specifications. We will only briefly describe these concepts because these ideas are well known; see, e.g., [Burstall and Goguen 77, Sannella and Tarlecki 88a].

2.1 Specifications

A *specification* is a finite presentation of a theory in higher-order logic. An uncommon feature of SPECWARE is that subsorts and quotient sorts can be defined using predicates and equivalence relations, respectively. For details of the particular logic used, see Appendix A.

2.1.1 Specification-Constructing Operations

Specifications can either be directly given (as a set of sorts, operations, axioms, etc.) or constructed from other specifications via the following operations (inspired by ASL [Wirsing 86, Sannella and Tarlecki 88a])

translate ⟨spec⟩ **by** ⟨renaming-rules⟩
colimit of ⟨diagram⟩
spec import ⟨spec⟩ ⟨spec-elements⟩ **end-spec**

"Translate" creates a copy of a specification with some elements renamed according to the given renamings; an isomorphism is also created between the original and the translated specifications. "Colimit" is the standard operation from category theory (see, e.g., [Mac Lane 71]); colimits are constructed using equivalence classes of sorts, operations, etc.

"Import" places a copy of the imported specification[1] in the importing specification; an inclusion morphism is also generated.

2.2 Specification Morphisms

A *specification morphism* (or simply a *morphism*) translates the language of one specification into the language of another specification in a way that preserves theorems. Specification morphisms underlie almost all constructions in SPECWARE.

2.2.1 Flavors of Specification Morphisms

The set of sorts given in a specification generates a free algebra via sort-constructing operations such as product, coproduct, etc. A specification morphism is a map from the sorts[2] and operations of one specification to the sorts and operations of another such that (1) the map is a homomorphism on the sort algebras, (2) the ranks of operations are translated compatibly with the operations, and (3) axioms are translated to theorems.

A presentation of a specification morphism in SPECWARE is a finite map from the declared sorts in the source specification to the declared or constructed sorts in the target specification, and from source operations to target operations, such that the map generates a specification morphism as described above.

Many flavors of morphisms can be defined for specifications, ranging from axiom-preserving presentation morphisms to logical morphisms between the toposes (theories) generated by the source and target specifications. The choice made in SPECWARE (declared sorts mapping to constructed sorts) is a pragmatic one, a compromise between simplicity and flexibility—morphisms are simple enough for use in putting specifications together, while flexible enough to model refinement.

2.3 Specification Diagrams

A morphism from A to B may be construed as indicating how A is a "part of" B. Thus, we can use morphisms to express a system as an interconnection of its parts, i.e., as a diagram. Formally, a *diagram* is a directed multigraph in which the nodes are labeled by specifications, and the edges by specification morphisms (in a multigraph, there can be more than one edge between any two nodes).[3]

2.3.1 Composition (Putting Specifications Together)

We can reduce a diagram of specifications to a single specification by taking the colimit of the diagram. The colimit of a diagram is constructed by first taking the disjoint union (coproduct) of all the specifications in the diagram and then the quotient of this coproduct via the equivalence relation generated by the morphisms in the diagram. The result will be

[1] Only one specification can be imported. A colimit is necessary if multiple specifications are to be imported.
[2] Here, we take "sorts" to mean all the sorts in the sort algebra.
[3] When convenient, we will treat a diagram as a functor from the category freely generated by its underlying graph to the category of specifications and specification morphisms.

a valid specification (i.e., the colimit exists) only if the sort algebra is free (this means that two structurally dissimilar sorts cannot be identified in a colimit).

Example 1. The specifications for topological sorting are shown in Fig. 1 (following Knuth [Knuth 68, pp. 258–265]). The problem of topological sorting is specified as an input-output relation. To specify this relation, we need the concepts of partial order and total order on some set of elements; these specifications are first put together via a colimit and then imported. The specification for partial orders contains a membership predicate and a less-or-equal predicate with appropriate axioms. The specification for total orders renames the partial orders specification and extends it with a totality axiom and a less-than predicate.

In the figure, the arrow labeled "d" is a definitional extension and the arrows labeled "c" are part of a colimit cocone.

3 Stepwise Refinement

The development process of SPECWARE is intended to support the refinement of a problem specification into a solution specification. Refinements introduce additional design detail, e.g., the transformation of definitions into constructive definitions, representation choices for data types, etc. SPECWARE's refinement constructs, introduced below, address three important aspects of refinement:

problem reduction: construction of a solution relative to some base;

stepwise refinement: sequential composition of refinements; and

putting refinements together: parallel composition of refinements.

3.1 Interpretations

The notion of refinement in SPECWARE is that a specification B refines a specification A if there is a construction which produces models of A from models of B [Sannella and Tarlecki 88b]. Specification morphisms serve this purpose because associated with every morphism $\sigma : A \to B$ there is a reduct functor $\_|_\sigma$ which produces models of A from models of B. Morphisms, however, are too weak to represent refinements which normally occur during software development. So, we use a more general notion, *interpretations*, which are specification morphisms from the source specification to a definitional extension of the target specification.

Definition 1 (*Interpretation*). An *interpretation* $\rho : A \Rightarrow B$ from a specification A (called *domain* or *source*) to a specification B (called *codomain* or *target*) is a pair of morphisms $A \to A\text{-}as\text{-}B \leftarrow B$ with common codomain $A\text{-}as\text{-}B$ (called *mediating* specification or simply *mediator*), such that the morphism from B to $A\text{-}as\text{-}B$ is a definitional extension.

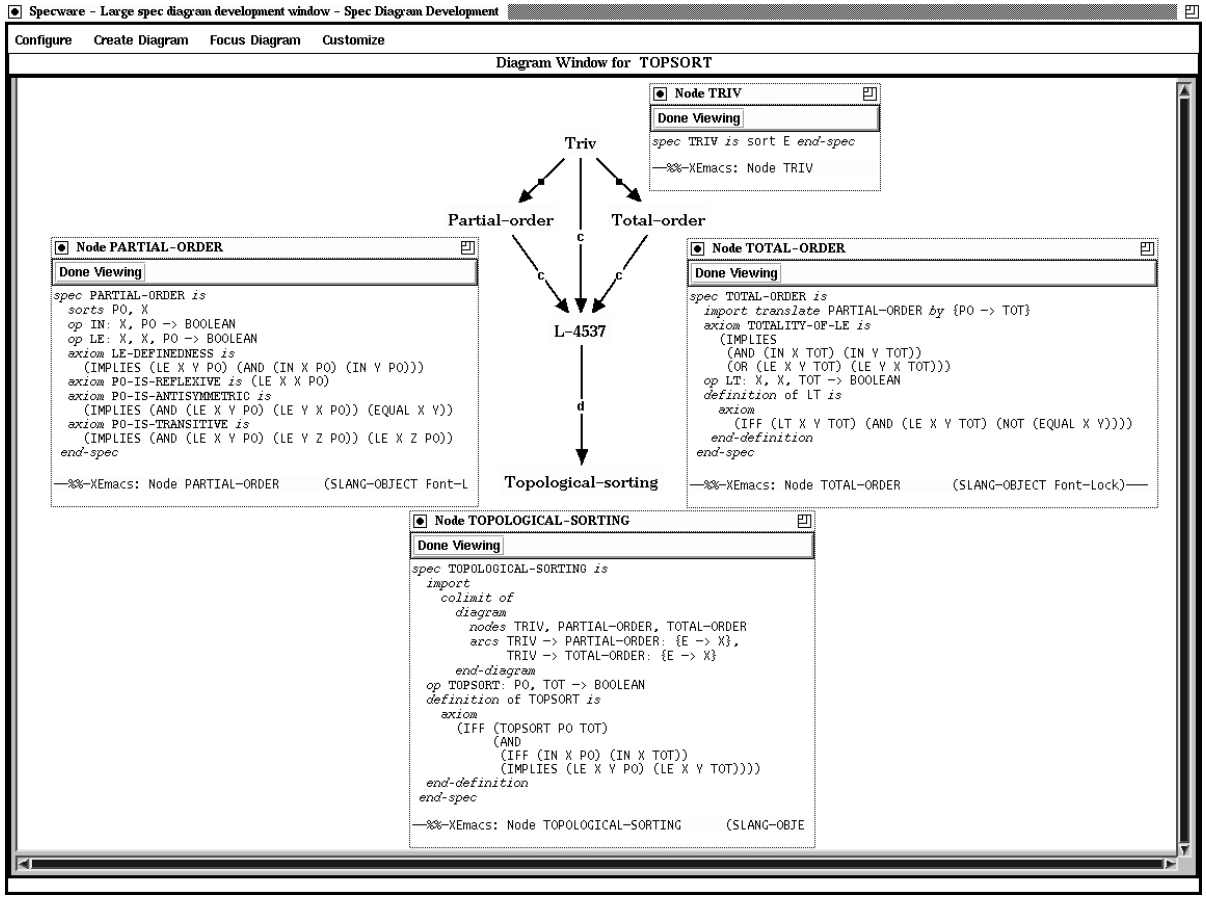


Fig. 1. Specification for topological sorting

Definition 2 (*Definitional extension*)**.** A morphism $S \to T$ is a *strict definitional extension* if it is injective and if every element of T which is outside the image of the morphism is either a defined sort or a defined operation. A *definitional extension* is a strict definitional extension optionally composed with a specification isomorphism.

In this case, we also sometimes say that T is a definitional extension of S. Definitional extensions are indicated in diagrams by $\xrightarrow{d}$.

A specification and any definitional extension of it generate the same topos (or theory). Hence, interpretations are generalized morphisms. Interpretations are a suitable notion of refinement because models of the source specification can be constructed from models of the target specification by first expanding them along the definitional extension and then taking reducts.

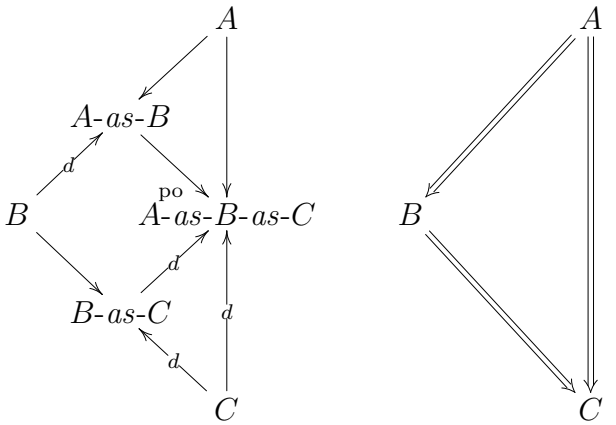
Example 2. We show in Fig. 2 an interpretation from total orders to sequences in which total orders are represented as a subsort of sequences: a sequence represents a total order if and only if it does not contain any duplicate elements. This subsort is defined in the mediating specification. Total-order operations are then defined on this subsort in terms of the underlying sequence operations.

In general, a source sort may be represented by a more elaborately constructed sort. For example, partial orders can be represented as a quotient of a subsort of graphs: to qualify as a representative, a graph must be acyclic (this is the subsort predicate), and two acyclic graphs represent the same partial order if their transitive closure is the same (this is the equivalence relation for the quotient sort).

Interpretations encompass and generalize the data type refinement introduced in [Hoare 72] and other similar schemes.

3.2 Sequential (Vertical) Composition of Interpretations

Given two interpretations $\rho_1 : A \Rightarrow B$ and $\rho_2 : B \Rightarrow C$ such that the codomain of the first is the domain of the second, their sequential composition $\rho_2 \circ \rho_1 : A \Rightarrow C$ is obtained as in the diagram below (the marking "po" indicates a pushout square).[4] We use the facts that definitional extensions are closed under composition and are preserved by pushouts.



[4] Diagrams are assumed to be commutative unless stated otherwise.

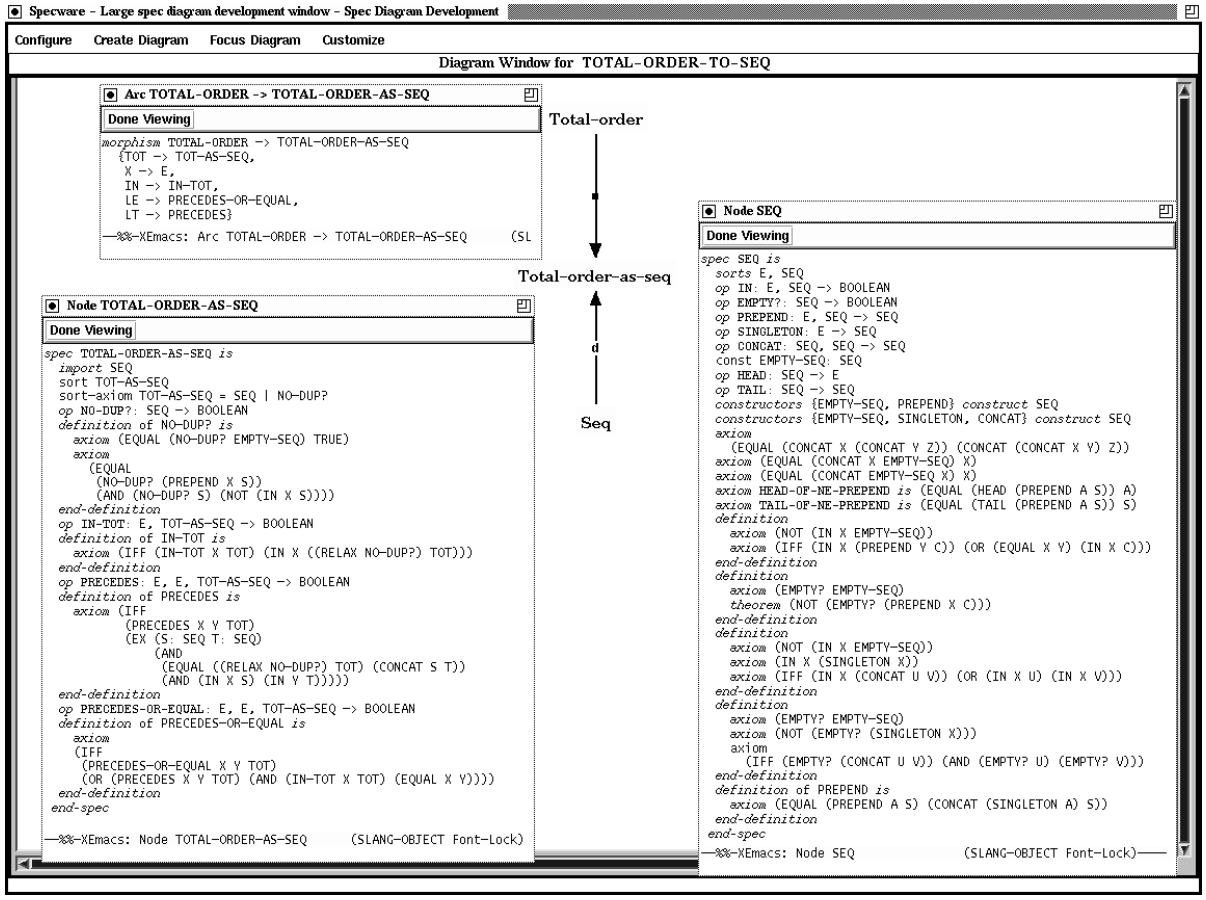


Fig. 2. An interpretation of total orders as a subsort of sequences

Sequential composition of interpretations facilitates incremental, layered refinement.

3.3 Algorithm Synthesis and Interpretation Construction

Algorithm synthesis plays two roles in the model of software development supported by SPECWARE:

- the creation of constructive definitions in interpretations, and
- the refinement of input-output relations sufficient to extract a constructively defined function.

Note that the definitions used in the mediating specification of an interpretation are not required to be constructive. As an example, see the definition of `PRECEDES` in the specification `TOTAL-ORDER-AS-SEQ` in Fig. 2. If we want to generate code corresponding to this operation, then we have to further refine this definition, with the goal of replacing the existential quantifier by an algorithm.

Similarly, the input-output relations used in a top-level specification are not usually functional. As an example, see the definition of the relation `TOPSORT` in the specification `TOPOLOGICAL-SORTING` in Fig. 1. If we want to find a function which satisfies this relation, we have to further refine the enclosing specification. This refinement can be guided by a hierarchy of algorithm theories which are used to impose additional structure on the specification. Details of this process can be found in [Smith 93, Smith and Lowry 90].

Algorithm synthesis is one of the creative parts of software development and can be used to construct basic interpretations which can then be composed. SPECWARE aids this by providing a scaffolding which takes care of the mundane details, thus letting the developer identify and focus on the creative part.

4 Putting Refinements Together

Just as a specification can be put together from smaller specifications, so can refinements of a specification be put together from refinements of component specifications. Formally, the various ways of constructing specifications generate a Grothendieck topology on the category of specifications and specification morphisms, and refinements form a sheaf with respect to this topology. Introductions to Grothendieck topologies and sheaves can be found in [Mac Lane and Moerdijk 92], [Artin et al. 72, Exposés I–IV]; an application to algorithm derivation and several computer science examples can be found in [Srinivas 93].

4.1 Theoretical Basis: A Sheaf of Refinements

Definition 3 (*A Topology for Specifications*)**.** We obtain a Grothendieck topology on the category of specifications and specification morphisms by defining a family of specification morphisms $\{ S_i \to S \}$ with common codomain to be a covering family if S is a definitional extension of the union of the images of the arrows in the family.

Definition 4 (*Image of a Specification Morphism*)**.** The image of a specification morphism $\sigma : S \to T$ is the specification consisting of all elements $\sigma(x)$ where x is any element of the source specification, e.g., sort, operation, theorem, etc.

To see that the topology above encompasses the specification constructing operations of Section 2.1.1, observe that a translation generates an isomorphism (which is a singleton covering family), and that a colimit specification is covered by its family of cocone arrows. The case of import can be reduced to that of colimit. However, it is useful to distinguish the case when the import morphism is a definitional extension; it then forms a (singleton) covering family.

Given any cover for a specification, a refinement for the specification can be constructed from refinements for the elements of the cover, provided the refinements are "compatible". This observation leads to a sheaf.

Definition 5 (*A Sheaf of Refinements*)**.** Assume a fixed specification B, the base specification. Define a functor $\mathcal{R} : \mathbf{Spec}^{\mathrm{op}} \to \mathbf{Set}$ by assiging to each specification S the set of all interpretations (refinements) from S to B, and to each specification morphism $m : S \to T$ the function which restricts an interpretation $\rho : T \Rightarrow B$ to an interpretation $\rho \circ m : S \Rightarrow B$. This functor is a sheaf with respect to the Grothendieck topology defined above.

The sheaf condition asserts that for every cover $\{ f_i : S_i \to S \mid i \in I \}$, every compatible family of interpretations $\{ \rho_i : S_i \Rightarrow B \mid i \in I \}$ can be uniquely extended to an interpretation $\rho : S \to B$ such that the restriction of ρ along any f_i is equal to ρ_i.

Informally, a family of interpretations $\{ \rho_i : S_i \Rightarrow B \mid i \in I \}$ is compatible if the member interpretations agree wherever the pieces of the cover overlap. In this case, an interpretation $\rho : S \to B$ can be constructed as the shared union of the given family of interpretations. The details of this construction will be omitted here, because the construction is similar to the parallel composition of interpretations described below.

4.2 Practical Realization: Diagram Refinement

Three factors prevent a direct realization of the sheaf-theoretic view of putting interpretations together presented in the previous section: (1) The compatibility condition is hard to check because pullbacks do not exist in general in the category of specification morphisms; (2) Equality of interpretations is hard to check; (3) It is unrealistic to assume that a single base specification (the refinement target) is given. Typically, we would like to assemble a target specification as we refine pieces of the source specification.

We handle (1) by using only those covers which are directly given by specification construction operations. In particular, a (finite) colimit explicitly indicates the shared parts among the components of a specification. (2) is handled by introducing interpretation morphisms, which explicitly indicate how one interpretation specializes another. We also use a strong equality for morphisms which can be checked syntactically; see Definition 6 below. (3) is handled by using diagrams in the category of interpretations and interpretation morphisms. A preliminary target specification can be assembled from the codomains of the

interpretations in a diagram. The target specification can be further modified by modifying the diagram of specifications that defines it.

We will describe these concepts below, finally obtaining a notion of refinement for diagrams.

Definition 6 (*Strong Morphism Equality*)**.** Two specification morphisms $\sigma, \tau : S \to T$ are equal if for each sort or operation $x \in S$, $\sigma(x) = \tau(x)$.

Definition 7 (*Interpretation Morphism*)**.** An interpretation morphism from an interpretation $\rho_1 : S_1 \Rightarrow T_1$ to another interpretation $\rho_2 : S_2 \Rightarrow T_2$ is a triple of specification morphisms such that the diagram on the right below commutes.

$$\begin{array}{ccc} S_1 & \Longrightarrow & T_1 \\ \Downarrow & & \\ S_2 & \Longrightarrow & T_2 \end{array} \qquad \begin{array}{ccccc} S_1 & \longrightarrow & S_1\text{-}as\text{-}T_1 & \xleftarrow{d} & T_1 \\ \downarrow & & \downarrow & & \downarrow \\ S_2 & \longrightarrow & S_2\text{-}as\text{-}T_2 & \xleftarrow{d} & T_2 \end{array}$$

Interpretations and interpretation morphisms form a category **Interp**. Another view of this category is as (a sub-category of) the functor category of functors from $\bullet \to \bullet \leftarrow \bullet$ to the category **Spec** of specifications and specification morphisms. Hence, colimits in **Spec** lift to colimits in the category of interpretations.[5]

Specifications, interpretations, and interpretation morphisms form a double category. That is, in addition to the obvious sequential/vertical composition of interpretation morphisms, there is also a parallel/horizontal composition of interpretation morphisms. The two compositions satisfy an interchange law: given six interpretations and four interpretation morphisms as shown on the left below, the equation on the right is true.

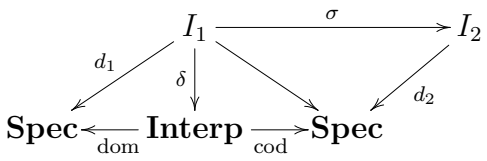
$$\begin{array}{ccccc} S_1 & \Longrightarrow & T_1 & \Longrightarrow & U_1 \\ \Downarrow{\scriptstyle \alpha_1} & & \Downarrow{\scriptstyle \beta_1} & & \\ S_2 & \Longrightarrow & T_2 & \Longrightarrow & U_2 \\ \Downarrow{\scriptstyle \alpha_2} & & \Downarrow{\scriptstyle \beta_2} & & \\ S_3 & \Longrightarrow & T_3 & \Longrightarrow & U_3 \end{array} \qquad (\beta_2 \bullet \alpha_2) \circ (\beta_1 \bullet \alpha_1) = (\beta_2 \circ \beta_1) \bullet (\alpha_2 \circ \alpha_1)$$

Now, given two specifications which are defined as colimits, a compatible family of interpretations can be given as a diagram of interpretations. It will be useful here to treat diagrams as functors.

Definition 8 (*Diagram Refinement*)**.** Given two diagrams of specifications $d_1 : I_1 \to \mathbf{Spec}$ and $d_2 : I_2 \to \mathbf{Spec}$, a diagram refinement $\langle \delta, \sigma \rangle : d_1 \to d_2$ is a pair consisting of a diagram of interpretations $\delta : I_1 \to \mathbf{Interp}$ with shape I_1 and a functor $\sigma : I_1 \to I_2$ between the two shapes such that the following diagram commutes (dom and cod are the obvious functors

[5] Definitional extensions are preserved by colimits.

which maps interpretations and interpretation morphisms to their domains and codomains, respectively).



Example 3. In Fig. 3, we show a refinement of the specification for topological sorting (shown in Fig. 1): the partial orders are refined to pairs of sequences (one listing the elements and another listing the ordering relation), and the total orders are refined to sequences (as shown in Fig. 2).

The components of the colimit which defines the import into the specification for topological sorting are refined in parallel. The vertical interpretations emanating from this diagram form a diagram refinement. Note that the target diagram has a shape which is different from that of the source diagram: the extra arrow in the target diagram is used to identify the sequences which represent the elements of the partial orders and the total orders (remember that topological sorting takes as input a partial order and produces a total order on the *same* set of elements).

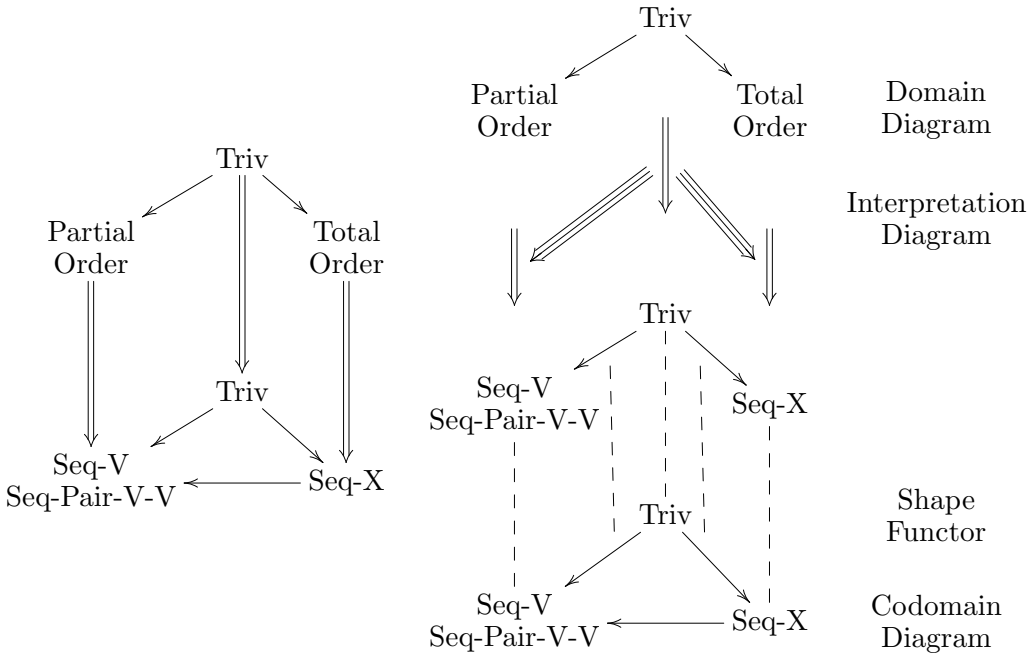
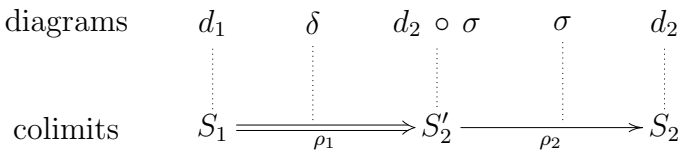


Fig. 3. Components of a diagram refinement

4.2.1 Parallel (Horizontal) Composition of Interpretations

As expected, a diagram refinement yields a refinement from the colimit of the source diagram to the colimit of the target diagram. Consider the diagram refinement $\langle \delta, \sigma \rangle : d_1 \to d_2$ above.

Let S_1 and S_2 be the colimits of the two diagrams. The colimit of the interpretation diagram δ is an interpretation $\rho_1 : S_1 \Rightarrow S_2'$ from S_1 to the colimit (say S_2') of the diagram $d_2 \circ \sigma : I_1 \to \mathbf{Spec}$. The colimit cocone $d_2 \dot{\to} S_2$ when composed with the shape morphism σ gives a cocone $d_2 \circ \sigma \dot{\to} S_2$. From this, we obtain a witness arrow $\rho_2 : S_2' \to S_2$. The composition $\rho_2 \circ \rho_1$ is the desired parallel composition of the diagram refinement $\langle \delta, \sigma \rangle : d_1 \to d_2$.

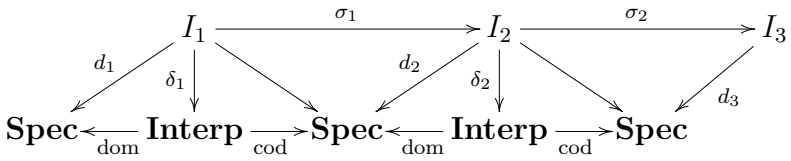


We will denote the parallel composition of a diagram refinement Δ by $|\Delta|$.

Example 4. In Fig. 4, we show details of the refinement of the specification for topological sorting. The figure illustrates both sequential and parallel composition of interpretations. As an example of sequential composition, partial orders are refined to pairs of sequences by representing them as graphs; the graphs are then represented as sets of nodes and sets of edges; then, these sets are represented as sequences. There are also several parallel compositions, e.g., the refinements of Set-of-Pair and TS-Import.

4.2.2 Composing Diagram Refinements

Diagram refinements can be composed by composing the individual interpretations which comprise them. Let $\langle \delta_1, \sigma_1 \rangle : d_1 \to d_2$ and $\langle \delta_2, \sigma_2 \rangle : d_2 \to d_3$ be two diagram refinements. We can juxtapose these as shown below.



Now, as shown below, we get two diagrams of interpretations with shape I_1, namely δ_1 and $\delta_2 \circ \sigma_1$, such that the codomains of the interpretations in the first diagram match with the domains of the interpretations in the second diagram. By composing the individual interpretations, we get another interpretation diagram with shape I_1. We will denote this horizontally composed diagram of interpretations by $(\delta_2 \circ \sigma_1) \bullet \delta_1$. The shape morphism for the composed diagram refinement is obtained by composing the individual shape morphisms, $\sigma_2 \circ \sigma_1 : I_1 \to I_2 \to I_3$. Thus, $\langle (\delta_2 \circ \sigma_1) \bullet \delta_1, \sigma_2 \circ \sigma_1 \rangle : d_1 \to d_3$ is the composition of the two diagram refinements we started with.

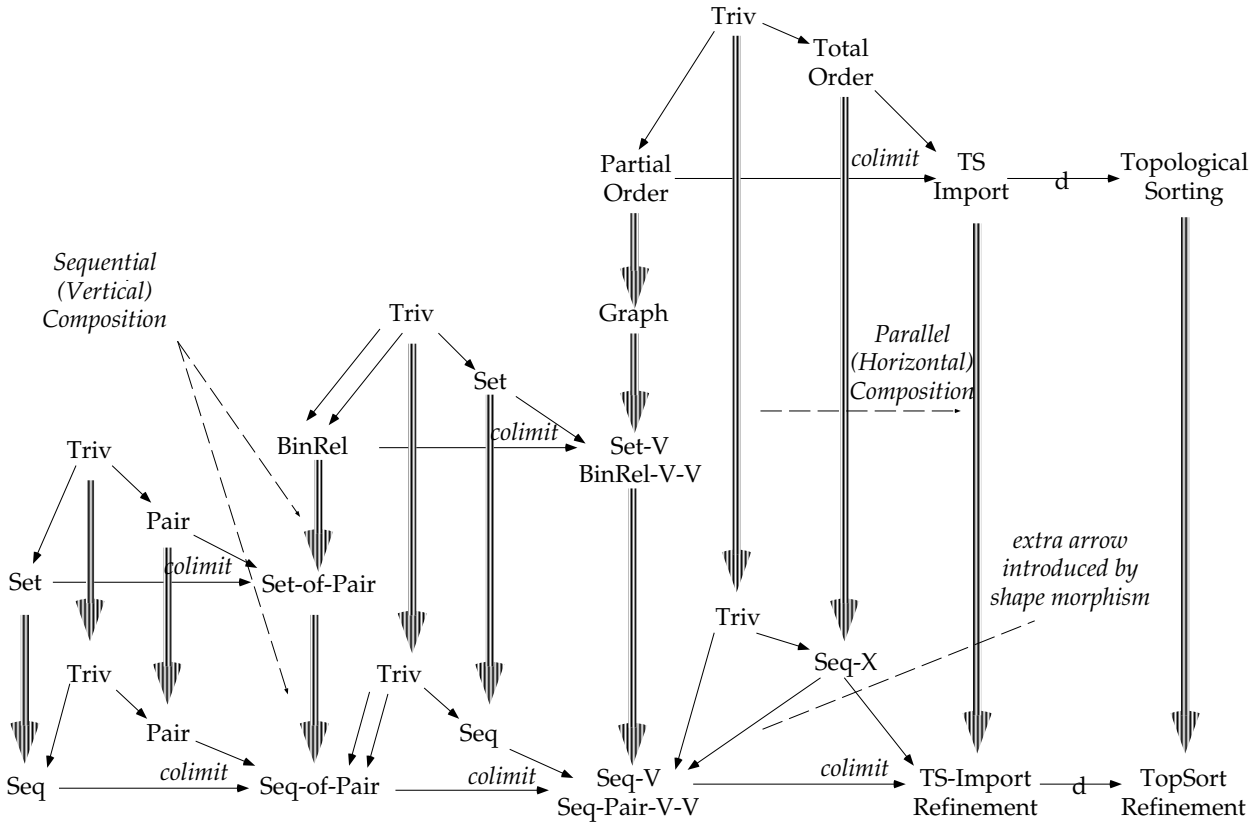
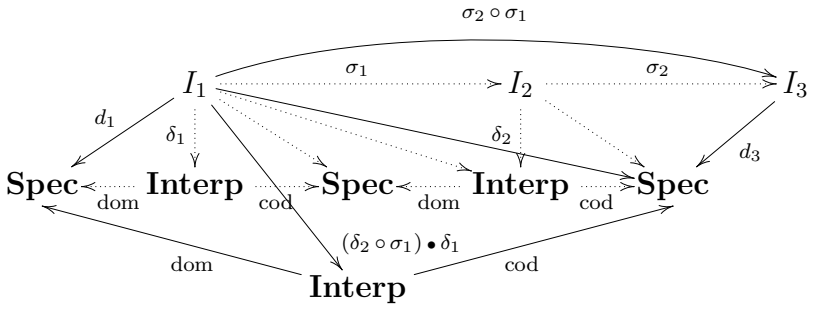


Fig. 4. Refinement of topological sorting



4.2.3 Compatibility of Vertical and Horizontal Interpretation Composition

If $\Delta_1 : d_1 \to d_2$ and $\Delta_2 : d_2 \to d_3$ are two diagram refinements which can be composed, then the distributive law satisfied by them is

$$|\Delta_2| \circ |\Delta_1| = |\Delta_2 \circ \Delta_1|.$$

This can be verified by straighforward diagram chasing (using the interchange law for interpretation morphisms). Thus, $|\_|$ is a functor from the category of diagrams and diagram refinements to the category of specifications and interpretations.

The distributive law above is a generalization of other such laws introduced in the literature. The law introduced by Goguen and Burstall [Goguen and Burstall 80] is too constraining to be practically useful. The law introduced by Sannella and Tarlecki [Sannella and Tarlecki 88b] uses parameterization and does not handle colimits; moreover, it is semantically oriented.

5 Putting Code Fragments Together

When specifications are sufficiently refined, they can be converted into programs which realize them. This involves a switching of logics. We use the theory of logic morphisms described by Meseguer [Meseguer 89]. We will confine our attention to entailment systems and their morphisms, rather than logics (which include models and institutions). Entailment systems are sufficient for the purpose of code generation.

5.1 Entailment Systems and their Morphisms

Definition 9 (*Entailment System*)**.** An entailment system is a triple $\langle \mathbf{Sig}, sen, \vdash \rangle$ consisting of

1. a category **Sig** of signatures and signature morphisms,
2. a functor $sen : \mathbf{Sig} \to \mathbf{Set}$ (where **Set** is the category of sets and functions) which assigns to each signature Σ the set of Σ-sentences, and to each signature morphism $\sigma : \Sigma \to \Sigma'$, the function which translates Σ-sentences to Σ'-sentences (this function will also be denoted by σ), and
3. a function $\vdash$ which associates to each signature Σ a binary relation $\vdash_\Sigma \subseteq \mathcal{P}(sen(\Sigma)) \times sen(\Sigma)$, called Σ-entailment,

such that the following properties are satisfied:

1. *reflexivity:* for any $\varphi \in sen(\Sigma)$, $\{\varphi\} \vdash_\Sigma \varphi$;
2. *monotonicity:* if $\Gamma \vdash_\Sigma \varphi$ and $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash_\Sigma \varphi$
3. *transitivity:* if $\Gamma \vdash_\Sigma \varphi_i$, for $i \in I$, and $\Gamma \cup \{ \varphi_i \mid i \in I \} \vdash_\Sigma \psi$, then $\Gamma \vdash_\Sigma \psi$;
4. $\vdash$*-translation:* if $\Gamma \vdash_\Sigma \varphi$, then for any signature morphism $\sigma : \Sigma \to \Sigma'$, $\sigma(\Gamma) \vdash_{\Sigma'} \sigma(\varphi)$.

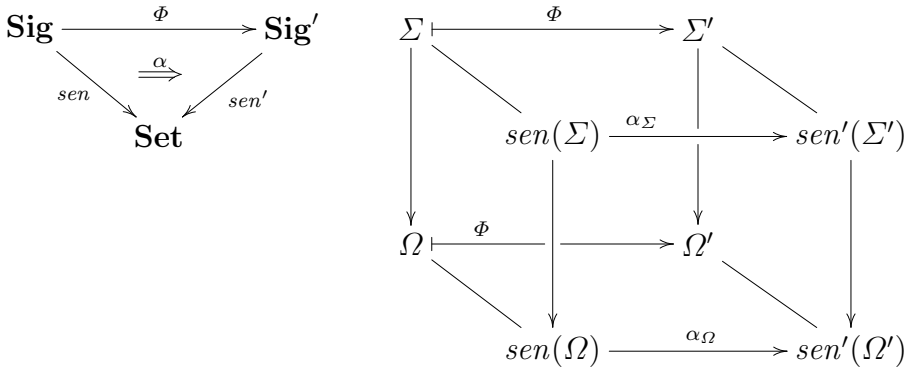
To map one entailment system into another, we map the syntax (i.e., signatures and sentences) while preserving entailment. Preservation of entailment represents the relevant correctness criterion for translating specifications from one logic to another. Note that this is similar to the correctness criterion for refinement within a single logic.

A simple way to map syntax is to map signatures to signatures, and sentences over a signature to sentences over the translated signature. If the former is a functor, the latter becomes a natural transformation.

Definition 10 (*Entailment system morphism—plain version*)**.** A morphism between entailment systems $\langle \Phi, \alpha \rangle : \langle \mathbf{Sig}, sen, \vdash \rangle \to \langle \mathbf{Sig}', sen', \vdash' \rangle$ is a pair consisting of a functor $\Phi : \mathbf{Sig} \to \mathbf{Sig}'$ which maps signatures to signatures and a natural transformation $\alpha : sen \dot{\to} sen' \circ \Phi$ which maps sentences to sentences such that entailment is preserved:

$$\Gamma \vdash_{\Sigma} \varphi \Rightarrow \alpha_{\Sigma}(\Gamma) \vdash'_{\Phi(\Sigma)} \alpha_{\Sigma}(\varphi).$$

We can visualize α and the naturality condition in the following diagrams.



Morphisms which map signatures to signatures are not flexible enough, especially for code generation. In general, it may be necessary to map built-in elements of one logic into defined elements of another, and vice versa. This can be realized by mapping signatures to specifications, and vice versa, or, in general, specifications to specifications.

However, morphisms which map specifications to specifications are too unconstrained. So Meseguer [Meseguer 89] proposes a general version of entailment system morphisms which map specifications to specifications "sensibly". We will use these morphisms but omit the detailed definition here.

5.2 Translating from Slang to Lisp

The specification language used in Specware is called Slang. We distinguish Slang because Specware may have multiple back-ends, Lisp, C, Ada, etc., each with its own logic.

We consider a sub-logic of Slang, called the *abstract target language* (for Lisp); there is one sub-logic for each language into which Slang specifications can be translated. We will denote this sub-logic by Slang^{--}. The sub-logic Slang^{--} is defined by starting with a set of basic specifications, such as integers, sequences, etc., which have direct realizations in the target language. All specifications which can be constructed from the base specifications, with the following restrictions, are then included in the sub-logic:

– for colimit specifications, only injective morphisms are allowed in the diagram;[6]

[6] For colimit specifications which can be construed as "instantiations" of a "generic" specification, the morphisms from the formal to the actual may be non-injective.

– all definitions must be constructive, i.e., they must either be explicit definitions (e.g., `(equal (square x) (times x x))`), or, if they are recursive, they must be given as conditional equations using a constructor set.

The goal of the refinement process is to arrive at a sufficiently detailed specification which satisfies the restrictions above.

The sub-logic SLANG^{--} will be translated into a functional subset of LISP. To facilitate this translation, we couch this subset as an entailment system, denoted LISP^{--}. The signatures of this entailment system are finite sets of untyped operations and the sentences are function definitions of the form

```
(defun f (x)
  (cond ((p x) (g x))
   ...))
```

and generated conditional equations of the form

```
(if (p x) (equal (f x) (g x))).
```

The entailment relation is that of rewriting, since theories in LISP^{--} can be viewed as conditional-equational theories over the simply-typed λ-calculus.

In Fig. 5, we show a fragment of an entailment system morphism from SLANG^{--} to LISP^{--}. Note, in particular, the translations from and to empty specifications. The set of sentences in the SLANG specification `INT` translates to the empty set; this is because integers are primitive in LISP. Similarly, the empty SLANG specification translates to a non-empty LISP specification; this is because some built-in operations of SLANG are not primitive in LISP.

5.2.1 Translating Constructed Sorts

There are numerous details in entailment system morphisms such as that from SLANG^{--} to LISP^{--}. We will briefly consider the translation of constructed sorts. Subsorts can be handled by representing elements of a subsort by the corresponding elements of the supersort. Similarly, quotient sorts can be handled by representing their elements by the elements of the base sort. Sentences have to be translated consistently with such representation choices: e.g., injections associated with subsorts (`(relax p)`) and the surjections associated with quotient sorts (`(quotient e)`) must be dropped. Also, the equality on a quotient sort must be replaced by the equivalence relation defining the quotient sort.

In Fig. 6, we show the representation of coproduct sorts by variant records. This translation exploits the generality of entailment system morphisms: a signature is mapped into a theory.

5.3 Translation of Colimits: Putting Code Fragments Together

If an entailment system morphism is defined in such a way that it is co-continuous, i.e., colimits are preserved, then we obtain a recursive procedure for translation, which is similar

SLANG^{--}	$\longrightarrow$ LISP^{--}
EMPTY	$\longmapsto$ spec SLANG-BASE is ops implies, iff (defun implies (x y) (or (not x) y)) (defun iff (x y) (or (and x y) (and (not x) (not y)))) end-spec
INT	$\longmapsto$ SLANG-BASE
spec FOO is import INT op abs : Int -> Int definition of abs is axiom (implies (ge x zero) (equal (abs x) x)) axiom (implies (lt x zero) (equal (abs x) (minus zero x))) end-definition end-spec	$\longmapsto$ spec FOO' is import SLANG-BASE op abs (defun abs (x) (cond ((>= x 0) x) ((< x 0) (- 0 x)))) end-spec

Fig. 5. Fragment of entailment system morphism from SLANG^{--} to LISP^{--}

```
spec STACK is                          ⟼  spec STACK' is
import INT                                 import SLANG-BASE
...                                        op size, E-Stack?, NE-Stack?
sort-axiom                                 (defun E-Stack? (s)
  Stack = E-Stack + NE-Stack                (= (car s) 1))
...                                        ...
op size  : Stack -> Int                    (defun size (s)
definition of size is                      (cond
axiom                                       ((E-Stack? s) 0)
(equal (size ((embed 1) s))                 ((NE-Stack? s)
       zero)                                 (1+ (size (pop (cdr s)))))
axiom                                      ))
(equal (size ((embed 2) s))                end-definition
       (succ (size (pop s))))              end-spec
end-definition
end-spec
```

Fig. 6. The representation of coproduct sorts as variant records

to that of refinement: the code for a specification can be obtained by assembling the code for smaller specifications which cover it.

The entailment system morphism from SLANG^{--} to LISP^{--} briefly described above does preserve colimits because of our restriction to injective morphisms. In general, this is true for most programming languages because they only allow imports, which are inclusion morphisms.

6 Related Work

SPECWARE builds upon a large body of work in formal specifications and program synthesis and transformation developed over the last two decades.

The design of SLANG, the specification language of SPECWARE, was inspired by Sanella and Tarlecki's [Sannella and Tarlecki 88a] and Wirsing's work [Wirsing 86] on structured algebraic specifications. Putting theories together via colimits was first proposed by Burstall and Goguen as part of CLEAR [Burstall and Goguen 77]. SLANG was further influenced by CIP [Bauer et al. 85, Bauer et al. 87] and OBJ [Goguen and Winkler 88].

SPECWARE adopts in a higher-order setting the notion of interpretations as refinements from Turski's and Maibaum's development in first-order logic [Turski and Maibaum 87]. SPECWARE could be construed as a realization of the design methodology espoused by Lehman, Stenning, and Turski, with the addition of parallel refinement composition [Lehman et al. 84]. The notion of parallel refinement composition described in this paper is different from the horizontal composition of parameterized specifications described by Sannella and Tarlecki [Sannella and Tarlecki 88b].

The explicit use of subsort and quotient sort constructions in SPECWARE connects data type refinement in an algebraic setting with Hoare's abstraction/refinement functions [Hoare 72] which also underlie the refinement found in VDM [Jones 86].

Our work is both similar and complementary to Bird's and Meertens' equational reasoning approach to program development [Bird 86, Bird 87]. Reasoning about commuting specification diagrams is equational reasoning at the specification level; Bird's and Meertens' equations are at the axiom level. Of course the two can happily co-exist.

Our framework for structured code generation is adopted from Meseguer's work on logic morphisms [Meseguer 89].

The direct impetus to the development of SPECWARE came from the desire to integrate several systems developed at Kestrel Institute over the last ten years, and the realization that they shared a common conceptual basis. These include the algorithm design system KIDS [Smith 90], the data type refinement system DTRE [Blaine and Goldberg 91], REACTO, a system for the development of reactive systems [Gilham et al. 89], and a synthesis system for visual presentations [Green 87]. An overview is presented in [Jüllig 93].

7 Conclusions

7.1 Summary

We presented the specification and refinement concepts of SPECWARE, a system aimed at supporting the application of formal methods to system development. Specware draws on theoretical work in formal specification and program synthesis as well as on experience with experimental systems over the past two decades. The development of SPECWARE continues; however, all concepts introduced here have been implemented. We have found the co-development of theory and implementation mutually beneficial.

The basic specification concepts of SPECWARE are specifications, specification morphisms, and diagrams of specifications and specification morphisms. The colimit operation takes diagrams of specifications to specifications.

The basic refinement notion is an interpretation, a morphism from a source specification into a definitional extension of a target specification. Interpretations are closed under sequential composition. To arrive at a notion of parallel refinement composition, we first observed that colimits and definitional extensions generate a Grothendieck topology on the category of specifications and specification morphisms, and that refinements form a sheaf with respect to this topology. Essentially this means that that given a specification diagram and an assignment of an interpretation to each node in the diagram one can construct an interpretation for the colimit of the given specification diagram, provided the compatibility condition holds: the interpretations assigned to the nodes must agree on shared parts.

The difficulty of checking the compatibility condition, among other reasons, prevented the direct application of this theory in practice. We instead developed diagram refinements as a practical realization; in diagram refinements the compatibility of interpretations is explicity ensured by the presence of interpretation morphisms.

7.2 Future Work

Current work includes adding to SPECWARE parameterized specifications and interpretations of parameterized specifications. This will lead to a vertical composition similar to that of Sannella's and Tarlecki's [Sannella and Tarlecki 88b] but to a different horizontal composition notion.

With the addition of parameterized specifications SPECWARE contains a set of primitives rich enough to allow for substantial experimentation. For this purpose we will recreate the algorithm design capabilities of KIDS in SPECWARE. We also expect the addition of code generation to other programming languages in addition to LISP.

A The Logic of Slang

The specification language used in SPECWARE is called SLANG. We distinguish SLANG because SPECWARE may have multiple back-ends, Lisp, C, Ada, etc., each with its own logic.

SLANG is based on higher-order logic, or higher-order type theory, as described in [Lambek and Scott 86]. However, unlike Lambek and Scott, we use classical logic (rather than intuitionistic logic) because the theorem prover currently used in SPECWARE is a resolution prover based on classical first-order logic (with some higher-order facilities).

Logically speaking, a SLANG specification is a finite collection of sorts, operations, and theorems (some of which are axioms). For pragmatic reasons, we have added sort-axioms (which are currently used to name sort terms), constructor sets (which are equivalent to induction axioms), and definitions (which are sets of axioms characterizing new operation symbols).

Every SLANG specification can be freely completed to a topos (see [Lambek and Scott 86, Section II.12] for a description of this construction). The objects in this topos are all sorts definable in the specification; the arrows are all definable operations (i.e., provably functional relations).

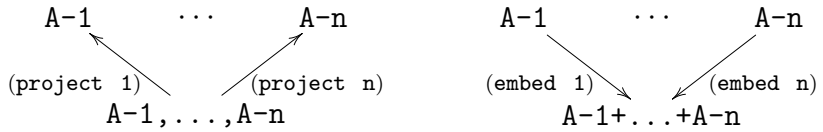
Built-in Constructs

The only sort which is built-in, i.e., is implicitly part of every specification, is `Boolean`. Along with this sort, the standard operations on it such as `true`, `false`, `and`, `or`, etc., and axioms characterizing them are built-in. The universal (`fa`) and existential (`ex`) quantifiers, and a polymorphic equality (`equal`) are also built-in.

Sort Constructors

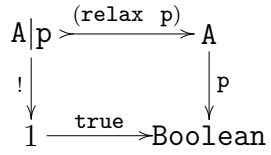
Lambek and Scott adopt a minimal set of sort constructors. While this is theoretically economical, we have chosen a richer set of sort constructors which arise in practice, especially in interpretations. We will use the generated topos to characterize these sort constructors; it is straightforward to generate the corresponding axioms.

N-ary products and coproducts. Given a set of n sorts, their product and coproduct are sorts which come equipped with the normal projections and embeddings, and characterized by the usual universal property.

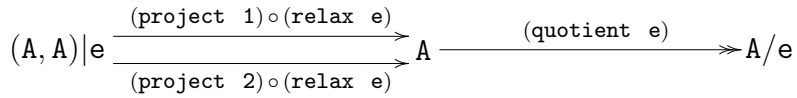


Function sorts. Given two sorts `A` and `B`, the function sort from `A` to `B`, written `A -> B`, satisfies the usual universal property and comes equipped with an evaluation operation, written `(<rator> <ap>)`, and an abstraction operation, written `(lambda (<args>) <body>)`.

Subsorts. Given a sort `A` and a predicate `op p: A -> Boolean` on this sort, the subsort of `A` consisting of those elements which satisfy the predicate, written `A|p`, and the induced injection are characterized by the following pullback diagram (1 is the terminal object, and ! denotes the unique arrow into it from `A|p`).



Quotient sorts. Given a sort `A` and an equivalence relation `op e:A,A -> Boolean` on this sort, the quotient sort consisting of equivalence classes of elements of `A`, written `A/e`, and the induced surjection are characterized by the following coequalizer diagram (`(A,A)|e` is the equivalence relation as a subsort of `A,A`).



Sort Axioms

Sort axioms are equations between sorts. Currently, these are restricted so that the left-hand side is a primitive sort (i.e., a sort which is not constructed using one of the sort constructors). Thus, in effect, sort axioms create new names for sorts. This keeps the sort algebra free, which is convenient for the type-checker. In the future, we may allow non-free sort algebras, and extend the type-checker to handle this.

Constructor Sets

A constructor set for a sort is a finite set of operations with that sort as the codomain. A constructor set is equivalent to an induction axiom. Here is an example.

```
constructors {zero, one, plus} construct NAT

axiom induction-for-NAT is
(fa (P) (implies
         (and (and (P zero) (P one))
              (fa (x y) (implies (and (P x) (P y))
                                 (P (plus x y)))))
         (fa (n) (P n))))
```

Note that a constructor set need not freely generate the constructed sort, i.e., the images of the constructors need not be disjoint. Additional axioms are necessary to force this.

Definitions

Definitions in SLANG are finite sets of axioms which completely characterize an operation. What this means is that to define a new operation `f: A -> B` in a specification S, there must be a formula `phi` with exactly two free variables `x:A` and `y:B` such that the relation specified by `phi` is provably functional in S:

```
(and (fa (x) (ex (y) (phi x y)))
     (fa (x) (implies (and (phi x y1) (phi x y2))
                      (equal y1 y2))))
```

Then S can be extended with the operation `f` together with the defining axiom

```
(iff (equal (f x) y) (phi x y)).
```

References

[Artin et al. 72]

ARTIN, M., GROTHENDIECK, A., AND VERDIER, J. L. *Théorie des Topos et Cohomologie Etale des Schémas*, *Lecture Notes in Mathematics,* Vol. 269. Springer-Verlag, 1972. SGA4, Séminaire de Géométrie Algébrique du Bois-Marie, 1963–1964.

[Bauer et al. 85]

BAUER, F. L., ET AL. *The Munich Project CIP, Volume I: The Wide Spectrum Language CIP-L*, *Lecture Notes in Computer Science,* Vol. 183. Springer-Verlag, Berlin, 1985.

[Bauer et al. 87]

BAUER, F. L., EHLER, H., HORSCH, A., MÖLLER, B., PARTSCH, H., PAUKNER, O., AND PEPPER, P. *The Munich Project CIP, Volume II: The Program Transformation System CIP-S*, *Lecture Notes in Computer Science,* Vol. 292. Springer-Verlag, Berlin, 1987.

[Bird 86]

BIRD, R. S. Introduction to the theory of lists. Tech. Rep. PRG-56, Oxford University Computing Laboratory, Programming Research Group, October 1986. Appeared in Logic of Programming and Calculi of Discrete Design, M. Broy, Ed., Springer-Verlag, NATO ASI Series F: Computer and Systems Sciences, Vol. 36, 1987.

[Bird 87]

BIRD, R. A calculus of functions for program derivation. Tech. Rep. PRG-64, Oxford University, Programming Research Group, December 1987.

[Blaine and Goldberg 91]

BLAINE, L., AND GOLDBERG, A. DTRE – a semi-automatic transformation system. In *Constructing Programs from Specifications*, B. Möller, Ed. North-Holland, Amsterdam, 1991, pp. 165–204.

[Burstall and Goguen 77]

Burstall, R. M., and Goguen, J. A. Putting theories together to make specifications. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* (Cambridge, MA, August 22–25, 1977), IJCAI, pp. 1045–1058.

[Gilham et al. 89]

Gilham, L.-M., Goldberg, A., and Wang, T. C. Toward reliable reactive systems. In *Proceedings of the 5th International Workshop on Software Specification and Design* (Pittsburgh, PA, May 1989).

[Goguen and Burstall 80]

Goguen, J. A., and Burstall, R. M. CAT, A system for the correct elaboration of correct programs from structured specifications. Tech. Rep. CSL-118, SRI International, Oct. 1980.

[Goguen and Winkler 88]

Goguen, J. A., and Winkler, T. Introducing OBJ3. Tech. Rep. SRI-CSL-88-09, SRI International, Menlo Park, California, 1988.

[Green 87]

Green, C. Synthesis of graphical displays for tabular data. Tech. Rep. SBIR.FR.86.1, Kestrel Institute, October 1987. Final Report for Phase I; Note: accompanying videotape.

[Hoare 72]

Hoare, C. A. R. Proof of correctness of data representation. *Acta Informatica 1* (1972), 271–281.

[Jones 86]

Jones, C. B. *Systematic Software Development Using VDM.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Jüllig 93]

Jüllig, R. Applying formal software synthesis. *IEEE Software 10*, 3 (May 1993), 11–22. (also Technical Report KES.U.93.1, Kestrel Institute, May 1993).

[Knuth 68]

Knuth, D. E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison-Wesley, Reading, Massachusetts, 1968.

[Lambek and Scott 86]

Lambek, J., and Scott, P. J. *Introduction to Higher Order Categorical Logic.* Cambridge University Press, Cambridge, 1986.

[Lehman et al. 84]

Lehman, M. M., Stenning, V., and Turski, W. M. Another look at software design methodology. *ACM SIGSOFT Software Engineering Notes 9*, 2 (April 1984), 38–53.

[Mac Lane 71]

Mac Lane, S. *Categories for the Working Mathematician.* Springer-Verlag, New York, 1971.

[Mac Lane and Moerdijk 92]

Mac Lane, S., and Moerdijk, I. *Sheaves in Geometry and Logic.* Springer-Verlag, New York, 1992.

[Meseguer 89]

Meseguer, J. General logics. In *Logic Colloquium'87*, H.-D. Ebbinghaus et al., Eds. North-Holland, 1989, pp. 275–329.

[Sannella and Tarlecki 88a]

Sannella, D., and Tarlecki, A. Specifications in an arbitrary institution. *Inf. and Comput. 76* (1988), 165–210.

[Sannella and Tarlecki 88b]

Sannella, D., and Tarlecki, A. Toward formal development of programs from algebraic specifications: Implementations revisited. *Acta Informatica 25*, 3 (1988), 233–281.

[Smith 90]

Smith, D. R. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16*, 9 (September 1990), 1024–1043.

[Smith 93]

Smith, D. R. Constructing specification morphisms. *Journal of Symbolic Computation, Special Issue on Automatic Programming 15*, 5-6 (May-June 1993), 571–606.

[Smith and Lowry 90]

Smith, D. R., and Lowry, M. R. Algorithm theories and design tactics. *Science of Computer Programming 14*, 2-3 (October 1990), 305–321.

[Srinivas 93]

Srinivas, Y. V. A sheaf-theoretic approach to pattern matching and related problems. *Theoretical Comput. Sci. 112* (1993), 53–97.

[Turski and Maibaum 87]

Turski, W. M., and Maibaum, T. E. *The Specification of Computer Programs.* Addison-Wesley, Wokingham, England, 1987.

[Wirsing 86]

Wirsing, M. Structured algebraic specifications: A kernel language. *Theoretical Comput. Sci. 42* (1986), 123–249. A slight revision of his Habilitationsschrift, Technische Universität München, 1983.