KES.U.96.7

Kestrel Institute

The Architecture of SPECWARE,™ a Formal Software Development System

by

Yellamraju V. Srinivas and James L. McDonald

August 1996

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

The Architecture of SPECWARE,™ a Formal Software Development System

Yellamraju V. Srinivas and James L. McDonald
Kestrel Institute
3260 Hillview Avenue
Palo Alto, CA 94304, USA
+1 415 493 6871
{srinivas,mcdonald}@kestrel.edu

August 1996

Abstract

SPECWARE is a tool that supports the modular construction of formal specifications and the stepwise and componentwise refinement of such specifications into executable code. SPECWARE may be viewed as a visual interface to an abstract data type providing a suite of composition and transformation operators for building specifications, refinements, code modules, etc. This view has been realized in the system by directly implementing the formal foundations of SPECWARE: category theory, sheaf theory, algebraic specification and general logics. The language of category theory results in a highly parameterized, robust, and extensible architecture that can scale to system-level software construction.

Keywords: Formal methods, software architectures, category theory implementation

Contents

1 Formal Software Composition

SPECWARE is an attempt to realize the best of formal methods research in a software development environment. It represents a synergy of decades of research in formal specifications—algebraic specification and general logics—and abstract mathematical theories originally invented for dealing with complex structures—category theory and sheaf theory.

Software development in SPECWARE is characterized by two tenets:

Description: We always deal with descriptions, i.e., a collection of properties, of the artifact that we ultimately wish to build. These descriptions are progressively refined by adding more properties, until we can exhibit a model or witness (usually a program) which satisfies these properties. Descriptions in SPECWARE are written in one of several logics.

Composition: We handle complexity and scale by providing composition operators which allow bigger descriptions to be put together from smaller ones. The colimit operation from category theory is pervasively used for composing structures of various kinds in SPECWARE. Besides composition operators, one needs bookkeeping facilities and information presentation at various abstraction levels. SPECWARE uses category theory for bookkeeping and abstraction.

1.1 Composition with Overlaps

Interesting interconnections of parts have overlaps. In SPECWARE, interconnections of components are represented by diagrams (a formal notion in category theory) of objects related by arrows which indicate the overlaps. The colimit operation produces a single object from the diagram by "gluing" the parts together along the indicated overlaps. Conversely, the diagram may be construed as a "covering" (a formal notion in topology) of the colimit object by parts. A composed object can be transformed or refined using transformations of the parts, provided these are "compatible" (a formal notion in sheaf theory), i.e., the sub-transformations agree where the parts overlap.

In Figure 1, we attempt to illustrate these concepts using geometrical figures. In the left half of the figure, two ellipses are glued along the shaded portions. Note that the composition inherits the shading of both parts. In the right half of the figure, the ellipses are transformed into other shapes, such that there is agreement on the overlap.

1.2 The Importance of Arrows

Arrows are fundamental in category theory and are used to represent the preservation of structure. An arrow indicates that the structure of the source object is also present in the target object, perhaps in a different form. In SPECWARE, arrows play a dual role. First, arrows are used to indicate how one object is a part of another in interconnection diagrams (see Figure 1). Second, they are used to indicate that the semantic properties of the source are preserved in the target. In the latter sense, arrows are used to represent refinement. The explicit use of arrows differentiates the composition mechanisms in SPECWARE from those used, for example, in object-oriented techniques.

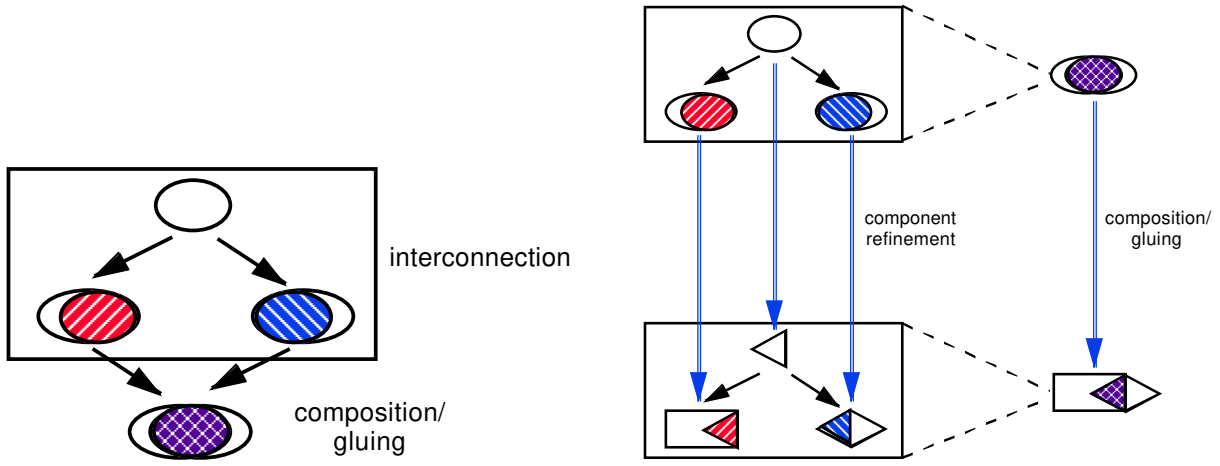

Figure 1: Composition and Refinement with Overlaps

2 The Specware Process Model

Specware is a tool that supports the modular construction of formal specifications and the stepwise and componentwise refinement of such specifications into executable code. Of course, this is an idealistic process; real software processes are usually iterative and evolutionary. However, all these processes require some basic infrastructure, and stepwise refinement already encompasses a significant portion of this infrastructure. Specware may be viewed as an extensible data type offering formal support for such processes.

2.1 Specifications and Refinement

Specifications in Specware are theories in a variant of higher-order logic called Slang [Srinivas and Jüllig 95, Lambek and Scott 86]. Specifications can be built modularly via specification-building operations such as import, translate and colimit. One specification can be refined into another (the latter being less abstract or more concrete) via an interpretation [Lambek and Scott 86, Turski and Maibaum 87]. An interpretation formally indicates how the types and operations of one specification are realized in terms of the types and operations of another specification.

Interpretations can be cascaded, thus resulting in stepwise refinement. Formally, specifications and interpretations form a category. Moreover, interpretations interact gracefully with the specification-building operations: a specification built from parts can be refined by refining its parts in a compatible way. Formally, interpretations form a sheaf with respect to the specification-building operations. We thus have a two-dimensional space of specifications related by the "part-of" relation in one dimension and the refinement relation in the other dimension.

```
spec SEQ is
  sorts E, Seq

  const empty-seq : Seq
  op prepend : E, Seq -> Seq

  axiom (not (equal (prepend e S) empty-seq))

  axiom (implies (equal (prepend a S) (prepend b T))
                 (and (equal a b) (equal S T)))

  constructors {empty-seq, prepend} construct Seq

  sort NE-Seq
  sort-axiom NE-Seq = Seq | non-empty?

  op empty?     : Seq -> Boolean

  definition of empty? is
    axiom (empty? empty-seq)
    axiom (not (empty? (prepend e S)))
  end-definition
...
end-spec
```

Figure 2: A Specification in Specware

2.1.1 Example

Figure 2 shows a fragment of the specification for sequences. A specification consists of a set of sorts (or types), a set of operations on these sorts, and a set of axioms which specify properties of the operations. Figure 3 shows an interpretation from total-orders to natural numbers. The mediating specification (NAT-with-lt) extends the target specification with a new operation and a definition for this new operation. The interpretation consists of the import morphism from NAT to NAT-with-lt together with the morphism from TOTAL-ORDER to NAT-with-lt. The latter morphism maps sorts to sorts and operations to operations such that the axioms of TOTAL-ORDER are satisfied (i.e., are theorems) in NAT-with-lt. Thus, this pair of morphisms builds a realization of the specification for total-orders in terms of the specification for natural numbers.

Figure 4 shows the piecewise refinement of a modularly built specification. The specification being refined describes the ingredients required for sorting: bags and sequences over elements which have a total ordering.

```
interpretation TOTAL-ORDER-to-NAT is
  mediator NAT-with-LT
  domain-to-mediator {Tot -> Nat,
                      lt -> nat-lt}
  codomain-to-mediator import-morphism

spec TOTAL-ORDER is
  sort Tot
  op lt : Tot, Tot -> Boolean

  axiom irreflexive is
    (not (lt x x))
  axiom transitive is
    (implies (and (lt x y) (lt y z)) (lt x z))
end-spec

   |
   | Tot -> Nat, lt -> nat-lt
   V

spec NAT-with-LT is
  import NAT

  op nat-lt : Nat, Nat -> Boolean
  definition of nat-lt is
   axiom lt-zero is
    (not (nat-lt x zero))
   axiom lt-succ is
    (iff (nat-lt x (succ y))
         (or (nat-lt x y) (equal x y)))
  end-definition
end-spec

   ^
   | import morphism
   |

spec NAT is
  sort Nat

  const zero : Nat
  op succ : Nat -> Nat

  constructors {zero, succ} construct Nat

  axiom succ-and-zero-are-disjoint is
    (not (equal (succ x) zero))
  axiom succ-is-injective is
    (implies (equal (succ x) (succ y)) (equal x y))
end-spec
```

Figure 3: An Interpretation in Specware

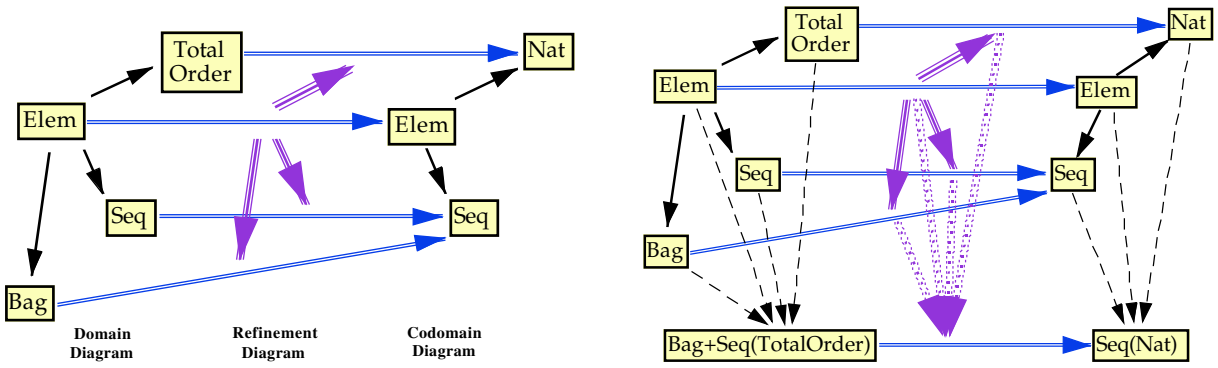

Figure 4: Diagram Refinement and Parallel Composition

2.2 Code Generation

A sufficiently refined specification can be transformed into executable code in programming languages such as Lisp and C++. This process is represented in SPECWARE as refinement into a different logic, i.e., programs are specifications too! Again, such inter-logic refinements can be composed. Figure 5 shows a fragment of the Lisp code in which bags (of bit-vectors) are represented as (equivalence classes of) lists.

2.3 Reasoning

Perhaps the main purpose of formality in software is to permit reasoning about specifications and their implementations. SPECWARE permits the use of multiple provers via the same mechanism of inter-logic refinements that is used for code generation. Reasoning within a specification requires a proof theory. A proof theory is a specification in the logic of a prover, a logic which is typically weaker than the higher-order logic that is used for specification. In a proof theory, axioms can be annotated with usage directives, e.g., left-to-right rewrite, and auxiliary information needed to control the prover, e.g., depth bounds, can be specified.

2.4 Logics and Morphisms

As observed above, one can write specifications in several different logics in SPECWARE. Abstractly, a logic consists of syntax and semantics. The syntax is usually given as a collection of signatures (sorts and operations) and a collection of formulas that are generated by the signatures. The semantics consists of a deduction relation between formulas (proof theory) and a collection of models and a satisfaction relation between formulas and models (model theory). The representation of logics and morphisms in SPECWARE is based on the description in [Meseguer 89].

Logic morphisms relate one logic to another by mapping the syntax while preserving provability and satisfaction. SPECWARE extends this notion to include modular construction of inter-logic refinements.

```
(in-package "CODE")

(load "/specware/2-0/library/lisp-code/list.lisp")
(load "/specware/2-0/library/lisp-code/slang-base.lisp")
(load "/specware/2-0/library/lisp-code/nat.lisp")
(load "/specware/2-0/library/lisp-code/char.lisp")
(load "/specware/2-0/library/lisp-code/string.lisp")
(load "/specware/2-0/library/lisp-code/seq.lisp")
(load "/specware/2-0/library/lisp-code/simple-vector.lisp")

(defun list-equal (l1 l2) (every #'sv-equal l1 l2))

(defun sl-remove (e l) (remove e l :test #'sv-equal))

(defun sl-remove-n (e l n) (remove e l :test #'sv-equal :count n))

...

(defun in-bag? (x s) (in x s))

(defun insert-bag (x s) (cons x s))

(defconstant empty-bag nil)

(defun remove-1 (x s)
  (cases
   (if (consp s)
    (if (not (sv-equal x (first s)))
     (return (cons (first s) (remove-1 x (rest s))))))
   (if (consp s) (if (sv-equal x (first s)) (return (rest s))))
   (if (endp s) (return s))))

(defun perm? (s1 s2)
  (cases
   (if (consp s1)
    (return (and (in (first s1) s2) (perm? (rest s1) (remove-1 (first s1) s2)))))
   (if (endp s1) (return (endp s2)))))

(defun in (x s)
  (cases
   (if (consp s) (return (or (sv-equal x (first s)) (in x (rest s)))))
   (if (endp s) (return nil))))

...

;;; Renamings

;;; equal_E -> Sv-Equal
;;; equal_Bag -> Perm?
```

Figure 5: Fragment of Lisp Code Generated by Specware

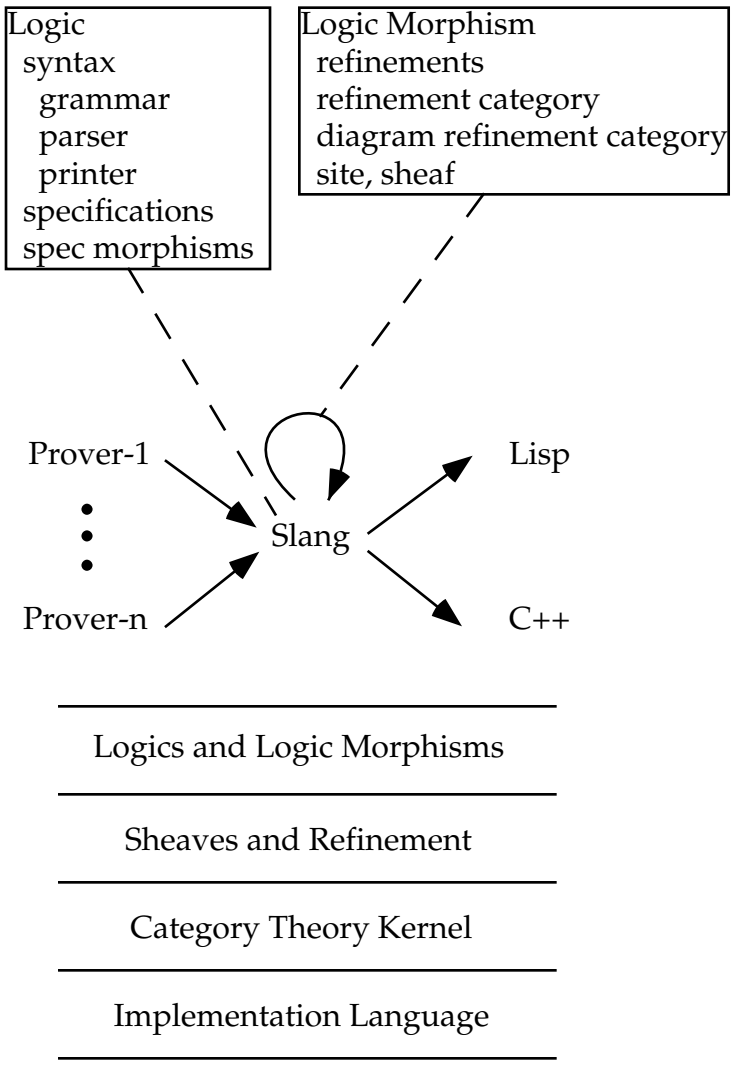

Figure 6: The Architecture of Specware

3 Architectural Support

Semantically, the SPECWARE workspace consists of a collection of logics related by inter-logic refinements (Figure 6). The logics supported by the current version of the system are the specification logic Slang, logics for (functional subsets of) Lisp and C++, and the logic of Kitp, a first-order resolution prover [Wang and Goldberg 94].

One of the goals of SPECWARE is extensibility. Rather than implement a specific set of logics, we chose to implement a generic set of capabilities that enables the addition of other provers, code generation for other languages, etc.

This generic set of capabilities forms a layered architecture consisting of

Implementation language: The language currently used in the SPECWARE implementation is an augmented version of Refine [REF85]. Other languages may be substituted

here, provided the features used by the other layers are available: a frame-like object representation with simple inheritance, constraint maintenance, and graphics.

Category theory kernel: This layer implements basic structures from category theory, including categories, functors, natural transformations, diagrams, and operations such as arrow composition and colimit.

Sites, sheaves, and refinement: This layers represents the notions of composition and refinement with overlaps. Sites augment categories with topologies which define the notion of covering an object by parts. Sheaves describe how a transformation of an object can be built from compatible transformations of its parts.

Logics and logic morphisms: This layer builds the structures necessary to represent different logics and refinements which span logics. Included here are generic facilities for handling syntax (parsing, printing, linking) and efficient representations of specifications, specification morphisms, and terms. Also included is an abstract representation of refinements from specifications in one logic to those in another.

The core of SPECWARE consists of a collection of abstract data types (e.g., categories, sheaves, etc.) organized via inheritance. A constraint maintenance system ensures syntactic and semantic well-formedness of instances of these data types.

3.1 Category Theory Kernel

Category theory was invented as an abstract language for describing certain structures and constructions which repeatedly occur in many branches of mathematics, such as topology, algebra, and logic. As opposed to set theory which is based on the membership relation and thus leads to the study of the internal structure of abstract entities, category theory takes morphisms or arrows as fundamental. Thus, in category theory, one studies the external properties of objects. To define an object, it is necessary and sufficient to describe its interaction (via morphisms) with all other objects.

The use of category theory results in parsimonious descriptions of entities, because of its abstract nature, and its focus on essential external properties. In recent years, it has found several applications in computer science, e.g., algebraic specification, type theory, programming language semantics, graph rewriting, automata theory, and even abstract machines based on categorical primitives. Introductions to category theory can be found in [Mac Lane 71, Pierce 91, Barr and Wells 90].

In SPECWARE, we exploit the ability of category theory to represent nested structures at multiple abstraction levels without losing the details. SPECWARE contains an extensive collection of basic structures used in category theory: categories, functors, natural transformations, shape categories, diagrams, diagram categories, internal categories, double categories, 2-categories, arrow categories, slice categories and functor categories. The implementation is derived from the presentation in [Freyd and Scedrov 90]: the somewhat formalist approach adopted in this presentation is ideally suited for an implementation.

To give a flavor of the implementation, we briefly describe the representation of categories. A category consists of a set of objects and a set of arrows connecting the objects, together

```
type category is
  objects : set(c-object)
  arrows  : set(c-arrow)
  identity-fn : c-object -> c-arrow
  composition-fn : seq(c-arrow) -> c-arrow
  colimit-fn : diagram -> cocone
  ...
end

type c-object is
  proto-object : specware-object
  parent-category : category
  identity-arrow : c-arrow
end

type c-arrow is
  proto-arrow : specware-object
  parent-category : category
  domain : c-object
  codomain : c-object
end
```

Figure 7: Representation of categories

with methods for composing arrows, building identity arrows, computing colimits, etc. Thus, a category is a structure with the signature shown in Figure 7. The objects and arrows in a category are themselves structured; the prefix "c-" is used to avoid confusion with the notion of "object" in the implementation language. Each object and arrow has a pointer to the parent category. The content of an object is contained in the proto-object slot (the content of an arrow is represented similarly); this allows SPECWARE entities to participate in several categories in different roles. For example, in the category of (Slang) specifications and morphisms, the proto-objects are specifications. The same proto-objects participate in the category of specifications and interpretations. On the other hand, the arrows in this category (interpretations) are the proto-objects in the category of interpretations and interpretation morphisms.

3.2 Semantic Constraints

Most SPECWARE entities, in addition to satisfying the types specified for their slots, must also satisfy semantic well-formedness constraints. For example, the parent-category of every c-arrow must be the same as the parent-category of its domain and codomain c-objects. Similarly, the sets of objects and arrows in a category must be compatible with the parent-category attribute. In SPECWARE, such semantic constraints are declaratively specified as

```
fa (arr : c-arrow)
parent-category(arr)
 = parent-category(domain(arr))

fa (cat : category, obj : c-object)
obj in objects(cat)
 <=> parent-category(obj) = cat
```

Figure 8: Example Semantic Constraints

shown in Figure 8. Simple constraints are automatically maintained by the system; for other constraints, repair actions must be explicitly indicated.

3.3 Building up Structure

To illustrate the use of category theory in building up complex nested structures, we describe the representation of diagrams and diagram refinements. These are defined generically, with an underlying category of specifications serving as the parameter. The composition operation for diagram refinements has a generic shell which invokes methods specific to the underlying category.

3.3.1 Diagrams as Functors

A diagram over a category is a directed multigraph whose nodes are labeled by objects in that category and whose arcs are labeled by arrows in that category. In other words, a diagram is a functor from a "shape category" (a category consisting of abstract nodes and arcs) to some other category.

For example, consider the left-half of Figure 4. The domain diagram has four nodes and three arcs; the nodes are labeled by specifications and the arcs by specification morphisms. The refinement diagram has the same shape, but now the nodes are labeled by interpretations and the arcs by interpretation morphisms (these indicate how one interpretation is a part of another).

3.3.2 Diagram Refinements

A diagram refinement is used to represent the piecewise refinement of an object. It is a compatible collection of refinements connecting a source diagram to a target diagram. Thus, a diagram refinement is an arrow, with the proto-arrow being a pair comprising a diagram of refinements and a shape morphism; the former has the same shape as the domain diagram, the latter relates the shape of the domain diagram to the shape of the codomain diagram.

The technical details of diagram refinements are presented in [Srinivas and Jüllig 95]; here, we just want highlight the nested structure. At the most abstract level, a diagram refinement is an arrow in the category of diagrams. The objects in this category are diagrams,

which are themselves arrows. Inside a diagram refinement is a refinement diagram and a shape morphism (again arrows). The refinement diagram is typically a diagram in a functor category. The objects in a functor category are again diagrams, and so on.

The parallel composition operator which glues together the component refinements in a diagram refinement recursively calls the colimit operation for these nested structures. This genericity allows a single mechanism to be used for the refinement of specifications to specifications, the refinement of specifications to Lisp, the refinement of specifications to C++, etc.

3.4 User Interface Design

The graphics for SPECWARE are designed as a visual interface to the abstract data type provided by the core of SPECWARE, which includes general structures from category theory such as objects, arrows, diagrams, etc., plus specific structures for specs, spec morphisms, etc. The operations in this abstract data type are the building blocks of the SPECWARE software process described previously: arrow composition, colimit, refinement via a cover, etc., hence we can capture not only a semantic design record but also a process record, both of which can be evolved.

The graphics interface is implemented using the triangle model described in [Coutaz 85]. An abstract structure sits at the apex of a (logical) triangle, whose left leg is a view mapping that structure into an object being presented, and whose right leg is a depiction rendering the abstract structure as graphical objects. This triangle induces a isomorphism carrying the abstracted structure back and forth between the internal object and its depiction on the screen. The advantage is modularization: the same abstract structure can be used to view many different kinds of objects, and the choice of depiction can be varied for a fixed view.

For example, a diagram or category can be viewed as a graph, and this graph could be depicted as boxes and arrows, a table of connections, etc. A diagram could also be viewed as a function from a shape category, and this function could be depicted as as a table of the pointwise mappings (essentially a view at right angles to the table above). A diagram refinement could be viewed as a prism depicted in three dimensions, with a view of the domain diagram in a foreground plane, and the codomain in a background plane (see Figure 4). As part of a larger structure, it could be viewed as an arrow and be depicted by a single line.

4 Parameterization and Extensibility

The combination of a frame-based representation and semantic constraints means the entities in SPECWARE are direct realizations of abstract data types for categories, functors, etc. Since the formal basis of SPECWARE is in terms of these data types, the result is an implementation which is surprisingly robust and extensible. To illustrate the abstractness and flexibility of the architecture, we briefly outline a few possible extensions to SPECWARE.

4.0.1 Addition of new back-ends

To add a code generation facility to another language (e.g., Ada), one needs to add the components which make up the new logic and the logic morphism from Slang to the new logic. This would include syntax (parser, printer, linker), a category of specifications and specification morphisms (generic representations of terms, morphisms, etc., can be reused here), a sheaf of inter-logic refinements. The graphics interface is completely reusable because it is based directly on the underlying categorical representation rather than on the specific logic.

The addition of new provers is similar.

4.0.2 Support for evolution

To support an evolutionary software process, one needs to capture design and process records. SPECWARE implicitly provides a design record in the form of a collection of refinements which relate the original specification to the final code. This implicit design record can be made explicit by adding a category of such objects to SPECWARE. By imposing structure on this category, the composition operators of SPECWARE become available.

Similarly, a process record can be captured. As observed above, the core of SPECWARE is an abstract data type whose operations are the building-blocks of the software development process. By formalizing and representing this data type within SPECWARE (i.e., a meta-theory of SPECWARE), this process becomes manipulatable (see, e.g., [Baxter 92]).

4.0.3 Software architectures

SPECWARE provides a rich vocabulary for describing components and interconnections, so it would be easy to add a category of architectures and their refinements. Although this category may have slightly different characteristics than categories of specifications, the composition and refinement machinery of SPECWARE is general enough to accommodate such new categories.

5 Lessons Learned

Over the past 4 years, SPECWARE has gone through several major revisions, with three significant beneficial events along the way.

The original version attempted (in retrospect too ambitiously) to implement a system for algorithm design based on category theory. That implementation, essentially a prototype, suffered from insufficient design and modularization. For example, a colimit operation was implemented, but the code was specific and ad hoc for one kind of colimit (specs and spec-morphisms). Furthermore, it conflated the notions of importation and parameterization, and included inessential and poorly characterized routines, e.g. code designed to choose names that would appeal to user's intuitions. Too much time was spent on "user-friendly" syntax and grammars. The net result was a system that was hard to understand, and hard to maintain or extend.

As a reaction to those problems, a "core" SPECWARE implementation was launched that attempted to build a clear, robust kernel one layer at a time, with each successive layer based on a clear design, and as simple and general as possible. Concomitantly, efforts to explicate the design of SPECWARE were accelerated. This radical change of course reduced the likelihood of immediate demonstration prototypes, but created the context for more stable and progressive long-term development.

The first implementation that followed was more modest in scope but much cleaner in design, and gave us confidence that a maintainable system could be built.

The second major event occurred a few months later when, as a result of continually modifying the core system to make it simpler and cleaner, it suddenly became apparent that a layer of "pure" category theory itself could be implemented on top of Refine and below the rest of SPECWARE. Since category theory is so well defined and documented, the implementation of that layer happened within days, and subsequently provided a excellent substratum for organizing and implementing what had been a diverse collection of objects and actions. It was at this point that the general design for SPECWARE really clicked into place.

For example, specification diagrams, which had been ad hoc data structures labeling nodes and arcs with specifications and morphisms, became functors from a shape category to the category of specifications and specification morphisms. Slight variations on that construction then gave us several other kinds of diagrams, and all of these were able to share the same graphics interface, for a large reduction in code size.

A third event occurred several months later when, based on the insights gained above, the entire system was adjusted to make the category/sheaf theory layers even more explicit and more complete. Since then, the core of SPECWARE has been exceptionally stable and robust.

Subsequently we have discovered that more peripheral aspects of SPECWARE benefit from the organizing principles given by category theory and sheaf theory. As we write, the connections of provers and graphics to the core are being reviewed and redesigned to clarify and capture as categorical constructions the essential structures that are preserved in passing from one domain to another. We hope to achieve a tight, natural, and extensible coupling that would be difficult to achieve by traditional software development.

References

[Barr and Wells 90]

BARR, M., AND WELLS, C. *Category Theory for Computing Science*. Prentice Hall, New York, 1990.

[Baxter 92]

BAXTER, I. D. Design maintenance systems. *Commun. ACM 35*, 4 (Apr. 1992), 73–89.

[Coutaz 85]

COUTAZ, J. Abstractions for user interface design. *IEEE Computer 18*, 9 (September 1985), 21–34.

[Freyd and Scedrov 90]

FREYD, P. J., AND SCEDROV, A. *Categories, Allegories*. North-Holland, Amsterdam, 1990.

[Lambek and Scott 86]

LAMBEK, J., AND SCOTT, P. J. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, Cambridge, 1986.

[Mac Lane 71]

MAC LANE, S. *Categories for the Working Mathematician*. Springer-Verlag, New York, 1971.

[Meseguer 89]

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

[Pierce 91]

PIERCE, B. C. *Basic Category Theory for Computer Scientists*. MIT Press, Cambridge, Massachusetts, 1991.

[REF85]

The REFINE^{TM} User's Guide, 1985.

[Srinivas and Jüllig 95]

SRINIVAS, Y. V., AND JÜLLIG, R. Specware:$^{\text{tm}}$ formal support for composing software. In *Proceedings of the Conference on Mathematics of Program Construction*, B. Moeller, Ed. Springer-Verlag, Berlin, 1995. Lecture Notes in Computer Science, Vol. 947.

[Turski and Maibaum 87]

TURSKI, W. M., AND MAIBAUM, T. S. E. *The Specification of Computer Programs*. Addison-Wesley, 1987.

[Wang and Goldberg 94]

WANG, T. C., AND GOLDBERG, A. KITP-93: An automated inference system for program analysis. In *Proceedings of 12th Conference on Automated Deduction*, A. Bundy, Ed. Springer-Verlag, Berlin, 1994, pp. 831–836. Lecture Notes in Artificial Intelligence, Vol. 814.