Kestrel

Kestrel Institute
3260 Hillview Avenue, Palo Alto, CA 94304
Ph/Fx: (650)493-6871

SPECWARE

Producing Software Correct by Construction

James McDonald
John Anton

March 14, 2001

Contents

1 Executive Summary **1**

2 Introduction **1**
2.1 Formal Software Development and Specware 3

3 Basic Principles **4**
3.1 Synthesis from Design 5
3.2 Orthogonality 7
3.3 Semantic Modularization 7
3.4 Taxonomies 8
3.5 Automation 9
3.6 Advantages 10

4 Category Theory **15**

5 Basic Categories **17**
5.1 Category of Specs and Spec-Morphisms 18
5.2 Category of Specs and Interpretations 21
5.3 Other Categories 22

6 Parameterized Specifications **22**
6.1 Illustration of problem 23
6.2 Solution 24
6.2.1 Contravariance 25
6.2.2 Example 25
6.2.3 Category of Pspecs 25

7 Hereditary Diagrams **26**
7.1 Diagrams as Designs 27
7.2 Lazy Colimits 29
7.3 Problems with old Approach 29
7.4 Promise of First Class Diagrams 29
7.5 Category of Diagrams 30
7.6 OS Example Revisited 30
7.7 Novel Kinds of Parameterization 31
7.8 Category of Diagrams 33
7.9 Category of Designs – Diagrams of Diagrams 34

1 Executive Summary

This report describes *Specware*^{TM} [1], a software development system that supports semantically precise compositions of specifications, and correctness-preserving refinements of those specifications into executable code.

Specware uses notions and procedures based on category theory and related mathematics to manipulate specifications. The net effect is a system that provides an orthogonal decomposition of software development into separate activities that can proceed independently, and which guarantees correctness when the results of those activities are combined. Specware has been used to synthesize multi-thousand line programs from requirement specifications.

2 Introduction

Kestrel Institute is developing the theory, technology and practice of formal software development; that is, the rigorous construction, using well-defined methods, of executable code that is known to meet some well-defined specification. In particular, three problems that commonly afflict large systems are addressed:

Assurance that code meets specification

Traditional software development systems have at best a rather loose coupling between specifications and code, and tremendous amounts of effort and ingenuity have been invested to address this fundamental shortcoming. Software analysis standards differ, but it is safe to say that most of the effort invested in programming, documentation, code reviews, test suites, monitoring, and customer support is concerned solely with assuring a correspondence between the code and its specification.

Flexibility and Productivity

A second problem with traditional approaches to complex software is that each product tends to be painfully composed from the efforts of diverse developers, and then lives on as a rigid standalone artifact.

[1] Specware is a trademark of Kestrel Development Corporation

It is hard to adapt components to work together correctly, it is hard to maintain such a system in light of changing specifications and environments, and it is hard to migrate all the specifications, documentation, code, test suites, etc. from one instance of a software family to the next.

Many of these problem are due to the large number of unstructured interdependencies that arise in code — a localized change in the requirements burgeons into an entangled nest of changes scattered throughout the code. Finding and effecting the changes requires a consideration of the system as a whole, so the cost of a change is proportional the the size of the entire system.

Performance

Significant performance issues with an existing product sometimes require radical design changes, but the turn-around cycle from such design changes to re-implementation is typically on the order of months or perhaps years. Traditional software development environments all but preclude systematic exploration of alternative design strategies tightly coupled to hard performance data.

Design and implementation decisions tend to be quite front-loaded and rigid. If end-product testing reveals unexpected performance issues, it can be difficult approaching impossible to revise an entire system to accomodate such belatedly discovered information, so everyone lives with the results until the next product cycle a year or more later.

Many methods have been proposed to overcome these three problems: procedural programming, standardized documentation, standardized development processes, object-oriented programming, etc. To a degree, these methods have been successful. However, large software systems developed today are still likely to be delivered late, cost more than predicted, deliver less than promised, require expensive maintenance, fail unpredictably, and fall prey to damaging exploits.

Kestrel has addressed these problems by developing technology that supports *rigorous and explicit modularity* in the *specification and development* of software components. The benefits are that a larger proportion of the development effort is directed to clarifying the specifications, that development can be factored safely into independent components guaranteed to combine

as expected, that the code produced is guaranteed to meet specifications precisely, and that changes in the requirements can be propagated through the specification and development process to produce changes in the code such that the cost of a change is proportional to the size of the change, not the size of the system.

2.1 Formal Software Development and Specware

Specware^{TM} is a *refinement*-based approach to software development that has been under development at Kestrel for several years now.

It is differentiated from most other specification/refinement systems in that its specification and refinement operations are based on a rigorous mathematical foundation, *category theory*, which promotes a high degree of confidence in their correctness. In particular, the semantics of refinement are such that provability is preserved: if a statement is a theorem in a specification, then it remains a theorem in refinements of that specification (taking into account possible renamings introduced by the symbol mappings). This is the basis of correctness-by-construction: code constructed through refinement from some specification is known to satisfy that specification because each refinement step preserves all of the properties of the original specification.

For a developer using Specware, software development proceeds something like this:

- First, the (functional) requirements on a software system are defined in an abstract specification. The specification is abstract in that: (i) it defines what the software is to compute–the input-output relation and related properties, and (ii) it is free of implementation details such as the choice of algorithms, or the use of particular concrete data types. In short, it says *what* the software should do, but *not how*.

- A typical specification is composed from component specifications. Dependencies between components are defined using *morphisms* or *interpretations* that syntactically map symbols from one component into symbols or terms in another component, with the crucial semantic restriction that the translations of theorems from the original spec must be provable theorems in the target spec, i.e., such morphisms must preserve theoremhood and provably show how to embed the source theory in the target theory.

- The abstract specification is refined into constructive form by introducing algorithms that are composed to meet the specification. These algorithms are introduced by constructing interpretations that map the formal specs for components into the formal specs for standard library algorithms.

- Abstract sorts are refined into concrete data types. Refinement may be performed by using morphisms or interpretations to show how a standard library data type can be used to represent a particular abstract data type.

- After the specification has been refined into a suitable form, standard library components (algorithms and data types) are converted into components in some executable programming language (e.g., C, C++, Lisp, or Java) using a code generator.

3 Basic Principles

A few basic principles have guided the development of Specware.

First and foremost is the principle of ***synthesis from design***, or alternatively, ***synthesis from specifications*** or ***synthesis by refinement***: code is generated by correctness-preserving refinements or transformations from abstract specifications, to provide ***correctness-by-construction*** in the final code. In what follows, specifications may also be referred to as specs, while the term *theory* is used to refer to the deductive closure of a specification — specs are the finite presentations of infinite theories.

The second principle is ***orthogonality***: specifications of different aspects of an application are designed, developed, and maintained as independently as possible from each other. Some theories describe a domain such as scheduling or graph layout, while others describe data structures such as strings or hash tables, while yet others describe algorithmic strategies such as divide-and-conquer or global-search. To date, Specware has focused on functional specifications that just define what a program should do, but potentially it could encompass theories to describe architectural aspects such as networking or client-server arrangements, performance aspects such as liveness or algorithmic complexity, or security issues such as privacy or robustness.

Third is the principle of ***semantic modularization***: specifications of any kind are designed as many small theories with strong semantic connec-

tions to each other. Each small theory is large enough to encompass the concepts and constraints comprising some meaningful software component, but is otherwise as small as possible, to maximize reusability. The connections used to glue theories together are required to maintain strict semantic compatibility, which can be thought of as behavioral type-checking (or even more informally as type-checking with a vengeance).

Fourth is the principle of ***taxonomies***: theories and techniques tend to form natural hierarchies from abstract to detailed versions, and such information can be captured and presented to developers as guides for software development. Furthermore, the structure of such a taxonomy may be exploited when progressing from designs using one node in the taxonomy to those using another node, especially if the second is a specialization of the first.

Fifth is the principle of ***automation***: as much as possible of the development process should be automated, to keep the developer's attention focused on just the crucial specification and design issues.

3.1 Synthesis from Design

Adherence to this principle can be characterized by various actions which are typically, but not necessarily, performed in the following order:

- Creation of semantically precise domain theories

  The developer in this step constructs theories that describe some realm in which problems will be posed. Typically this will be done once in consultation with experts in the domain area, and the (perhaps substantial) effort expended here will provide the basis for whole families of applications.

  These specifications should be domain-specific, but computationally abstract. Data structures at this level will typically be in terms directly meaningful to the customer: schedules, networks, databases, etc. Procedures and predicates at this level will be characterized primarily by their abstract functionality: feasibility of a schedule, throughput of a network, latency in a database, etc.

- Creation of semantically precise specifications

  The developer in this step constructs specifications that describe precisely what is desired, and no more: irrelevant details and implementation details should be left unspecified. The developers of normal applications should spend the vast majority of their time doing this, in tight consultation with customers.

  As the specifications are posed in terms of the domain model developed above, they should be domain-specific, but computationally abstract.

  Note that in spite of their computational generality, specifications at this level will include axioms stating precisely what is demanded of an application: fairness, liveness, no late deliveries, no single-point failure nodes, no races, optimality, etc. These are the crucial properties that the final code will be guaranteed to maintain.

- Refinement of algorithms

  Given an abstract algorithm such as sorting, there are many ways to refine it into a specific algorithm such as bubble-sort, mergesort, quicksort, etc. Specware supports such refinements in a manner which guarantees that all stated properties of the abstract algorithm will be maintained in the refined versions. For many applications, the precise refinement will hardly matter, while for high performance applications the developer should have the opportunity to make crucial decisions. Experimentation can be facilitated by providing taxonomies of algorithms to choose among and by automating subsequent steps towards code generation.

- Refinement of data structures

  The story here is almost exactly analogous to that for algorithms.

  Given an abstract structure such as a schedule, there are many ways to refine it into maps, sequences, arrays, etc. Again, Specware supports such refinements in a manner which guarantees that all stated properties of the abstract operations will be maintained in the refined versions.

  For many applications, very little time or expertise will be needed here, as much of this refinement can be fully automated. However, for high performance applications, there is room here for the developer to make

crucial decisions that may affect performance. Experimentation with respect to these crucial decisions is facilitated by providing taxonomies of refinements to choose among and by automating subsequent steps towards code generation.

- Generation of code

  Once the original abstract specifications have been refined to detailed specifications using specific data structures and algorithms, the final step of generating appropriate code should be fully automatic.
  Specware supports fully automatic generation of Lisp, C and C++ code, with Java a likely next target, and in principle Specware could target almost any imaginable target language: assembly, hardware, database, markup, etc.

3.2 Orthogonality

The essence of this principle is that the properties of domains, architectures, data structures, algorithms, etc. can be described and maintained independently for each such realm.

Adherence to this principle enormously increases potential for reuse. Given 10 domains, 10 architectures, 100 algorithms, and 100 data structures, one potentially can generate about a million applications. The same effort invested in traditional programming might yield a few hundred applications.

3.3 Semantic Modularization

The essence of this principle is that the specifications for any given aspect should be composed of many small theories, with precise semantic glue connecting them.

- Many small theories

  Small theories that describe a particular aspect of computing are more likely to be described completely, correctly and elegantly, and are vastly more likely to be reused.

  The relevant behaviors for sets, maps, strings, etc. can be described in theories that one can read and almost immediately understand. Moreover, one can then (both mentally and mechanically) encapsulate them and consider sets of integers, maps from accounts to customers, etc.

In short, gluing lots of small theories together makes an overall structure that is vastly more comprehensible, reusable, and reliable than having one monolithic theory.

- Semantically precise glue — spec morphisms

  The way in which theories are glued together is a crucial aspect of Specware. Traditional software paradigms use relatively weak constraints that at best use strong type checking to verify that one component can speak intelligibly to another. Without explicit semantics, however, there is no way to adequately constrain the behavior of modules when they interact. A routine that expects to sort using some binary function may require that such a function be a total order, but in traditional paradigms will be happy to link with a binary relation that produces random boolean values!

  In Specware, the morphisms that connect specifications are required to translate all theorems of the source specification into conjectures that are provable in the target specification. In other words, morphisms are required to embed the entire source theory in the target theory, or in yet more words, any possible model for the target theory must obey all the rules posited by the source theory.

  This ensures that whenever two specifications are connected by a morphism, in whatever context, they will be semantically compatible. This property is heavily exploited by high-level operations in Specware.

3.4 Taxonomies

Various aspects of the software environment can be captured in taxonomies: domain models, architectures, data structures, algorithms, etc.

Having such taxonomies can reduce much of the work in application development to selecting the appropriate nodes in those taxonomies. For example, one might request a sorting algorithm (from a domain taxonomy) that is implemented for strings (from a data taxonomy) using divide-and-conquer (from an algorithm taxonomy) on a uniprocessor (from an architecture taxonomy), and then merely be left with the manual selection of simple decomposition (yielding mergesort) or simple composition (yielding quicksort). Finally, the choice of target implementation language can come from such a taxonomy.

3.5 Automation

Since the Specware development process uses many highly detailed specifications, there is a great need (and fortunately there are great opportunities) for automating much of the formal detail. Such automation can apply locally or globally.

Many opportunities for automation can be characterized by the fact that a partial expression or partially expressed operation can be expanded into a unique complete expression or operation.

- Type Inference

  The algebraic specifications used in Specware use strongly typed expressions, but automatic type inference can infer the appropriate types in the vast majority of cases, and resolve overloaded symbols. This has been in Specware from the beginning.

- Type Coercion

  Often one wishes to apply an operation defined on one sort to an object belonging to a subsort, or vice versa. Automatic type coercion can insert the appropriate conversions and when necessary posit the appropriate proof obligations for this to be meaningful. Type coercion is available in metaSlang, a new version of Specware now under development.

- Morphism Completion

  Once a few source symbols in a morphism have been mapped, it may be possible to automatically deduce the mappings for all the remaining symbols. If there is only one possible morphism, it may be possible to deduce it with no initial mappings. This is available in currently released versions of Specware.

- Theorem Provers

  In some contexts, a theorem prover may be used to answer queries or find witnesses, which can then be automatically exploited. The morphism editor, for example, allows the user to request unskolemization and witness-finding to deduce the appropriate definition for a freshly defined target operation to be used as the target for some source operation.

- Colimits

  There are efficient algorithms for computing colimits of the kinds of diagrams used by Specware. The objects and arrows those diagrams refer to may be arbitrarily large or complex, allowing a vast amount of detailed activity to be subsumed into one semantically well-defined step.

Other opportunities for automation arise by having appropriate high-level operations that make large useful steps.

- Refinements

  If the user has specified a course of action such as refinement to code, it may be possible in many cases to automatically decompose the abstract specifications into smaller pieces, refine those recursively, and then automatically compose the results.

- Transformations

  Some actions such as finite-differencing may be described abstractly as pattern matching followed by extensive rewriting.

- Tactics

  Some sequences of actions are canonical, in which case users may wish to define their own tactics to perform them. The new MetaSlang version of Specware is written almost entirely within MetaSlang itself, which should facilitate the expression of such tactics in MetaSlang and their fairly direct introduction by the user at runtime.

The list above is not exhaustive. In general, we exploit most opportunities we find to introduce (semi-)automation. Overall, we strive for each user interaction to perform some meaningful task.

3.6 Advantages

There are several advantages to following the above principles, and these appear in all aspects and stages of software development.

- Correctness

  Most importantly, developers spend the majority of their time where it matters most — correctly describing applications and thinking about appropriate implementation strategies.

  Classical studies, such as those described in *Software Engineering Economics* by B. Boehm[3], show that the effort required to fix bugs grows exponentially with each step of the software life cycle. To quote from page 40: "These factors combine to make the error typically 100 times more expensive to correct in the maintenance phase on large projects than in the requirements phase.", with the footnote "The total economic impact of leaving errors to be found after the software has become operational is actually much larger, because of the added operational costs incurred by the error."

  An experiment run by James Widmaier at NSA compared the use of Specware technology with the use of a facility rated at level 4 using SEI's Capability Maturity Model [8]. The results were published as "A Comparison Between Standard and Formal Mathematical Software Development", Xin Huang's Master of Science Thesis, University of Maryland Department of Nuclear Materials and Reliability Engineering, 1999 [Huang99].

  The Specware and CMM4 teams were each given the same time and money, the same initial requirements, and the same access to the customer, and each was motivated to perform well. The team at Motorola using Specware spent 32% of its time in the requirements phase, uncovering several problems there, whereas the CMM4 team spent merely 11% of its time there, and found far fewer problems. The application produced by the Specware team was judged by audited independent third parties as having a 77% reliability, compared to 58% for the CMM4 team. Moreover, fixing two well-localized design errors would quickly have raised the Specware score to 98%, whereas no simple correction would substantially improve the CMM4 score. This provides hard evidence that the Specware approach encourages developers to focus on specifying requirements, and that such time is very well spent.

- Performance

  There are three major components to maximizing the performance of a program: choose the best data structures and algorithms, apply high

level optimizations to reorganize the code, and fine tune the resulting code sequences.

- Algorithmic performance

  As any instructor for software 101 will tell you, by far the biggest gains in performance come from selecting the appropriate algorithms and strategies, not from fiddling with instruction sequences or low-level details. The former may help a program run thousands (or trillions!) of times faster, the latter might gain a factor of two or so. Primarily because Doug Smith focused on the proper class of algorithm, the TPFDD schedulers he developed at Kestrel were orders of magnitude (from 25 to 250 or more times) faster than competitive schedulers that had been developed using traditional technology from the AI and operations research communities. [13]

- High level optimizations

  The next largest gains in performance come from somewhat complex code transformations such as finite-differencing or memoization, in which the calculation of redundant computations is avoided by the introduction of new data structures. Recognizing the sites for and applying such transformations is a forte of KIDS, Specware's predecessor. Some of this capability has been added to Specware, and more will be.

- Fine tuning, constraint propagation

  When code is synthesized mechanically from specifications, optimizations can be applied aggressively and systematically in a manner that would make manually developed code too brittle to maintain. With a system of automated refinements where each step provably maintains correctness, each line of code can take into account the entire semantic context it runs in to eliminate impossible paths, optimize orderings, delete redundant verifications, etc. These kinds of optimizations are natural extensions of those done by optimizing compilers for traditional languages. Again, KIDS excels here, and some of this capability has been added to Specware, while more will be.

In short, making algorithm refinement a separate and decomposable activity provides opportunities both large and small to increase performance.

- Reuse

  Specifications are vastly more reusable than implementations.

  By making specifications (as opposed to implementations) the primary repository of information about an application, this type of reuse is maximized.

  A generic specification of maps can be exploited in more contexts than any specialized implementation using arrays, bit vectors, hash tables, association lists, etc.

  Generic strategies such as divide-and-conquer or global search can be used in countless contexts once they have been abstracted from the entangling details of particular programs.

  Domain specifications for something like scheduling can be combined with various problem specifications to yield whole product lines. Domain specifications for something like inventories can be used to synthesize software for purchasing, warehousing, distribution, sales, tax issues, etc. A specification of an aircraft's physical layout could be used in one context to generate code for computing sound propagation, in another for programming machines to fabricate parts, and in another to generate code for laying out movable components.

- Reliability

  Automating the refinement to code eliminates coding errors.

  Each step in the Specware refinement process is constrained to preserve correctness, using theorem provers if necessary to verify this. Thus any property stated in the original specification (e.g. "a plane will never be scheduled to depart without a full well-rested crew available") can be maintained in the final program. No misplaced comma or semi-colon, stack overflow, or too-clever coding trick will lead to a violation of that requirement.

  This does not eliminate all bugs from the application; for example, the intended requirement above may really have been "a plane will never be scheduled to depart without a full well-rested crew aboard". However, it makes it possible for the testing effort to focus on testing the correctness of the specification, which is a much smaller and focused task, and which occurs with respect to language that is directly meaningful and relevant to the client. Note also that tracking

the detection and correction of problems at that level provides a basis for highly proportionate and appropriate management of resources devoted to development, testing, and maintenance.

- Time to Market

  Automating the refinement to code makes the implementation phase thousands of times faster than manual coding. This creates a very tight cycle of generating code, assessing a particular implementation of an application, modifying the specification of the application in light of experience with it, and generating new code. Once the domain model for a family of applications has been developed, the time for the generation of new applications meeting revised specifications might be measured in minutes instead of months. In fact, we have demonstrated the generation of alternative schedulers to meet user-selected criteria in a manner of minutes.

  Of course, no lunch is completely free. The original development of the domain theories for a completely new application area can take a significant amount of time, perhaps years. However, each new application area is likely to find more and more off-the-shelf specifications exploitable from previously developed applications, so in the long run, the time to create such domain models will likely be a modest investment proportional to the novelty of the domain.

- Maintenance

  Having just the code for an implementation is like having just the tip of an iceberg, or perhaps just the tail of a tiger. The vast bulk of the relevant information needed to revise an application lies elsewhere. Specifications, on the other hand, can capture most or all of that information. Even after the original implementors have long departed, new developers can look at the original specifications in light of new customer demands, quickly make adjustments, and quickly generate new applications. Specifications provide an invaluable "corporate memory", and refinement technology makes revising code a relatively painless task.

- Documentation

  Invariably, the time spent developing a precise specification leads to a much more detailed understanding of the customer's needs, by both

the developer and the customer. Since the specifications are close to the customer's terminology, recasting them in meaningful natural language should provide excellent documentation. Among other things, such documentation should illuminate often unstated assumptions and (using semantic inference capabilities) perhaps hard-to-foresee consequences, making the user's expectations of the application more meaningful, vivid, concrete, and useful. This is an activity that not yet been explored by Kestrel.

Assuming we want synthesis from design, there is still the question of how designs and refinement steps are structured. The short answer is that we want specifications to be highly modular and confined to specific aspects. There should be many small theories connected in semantically precise ways to compose larger theories, and the composition should be hierarchical, encapsulating detail.

4 Category Theory

Category theory is the obvious choice for the organizing principle of Specware. It is the pre-eminent organizing principle for modern mathematics, and by building upon such a fundamental mathematical basis, Specware can take advantage of (and contribute to) a large and growing body of work by the mainstream communities in mathematics, computer science, and even physics.

At a very abstract level, category theory is concerned with the way in which properties of objects are preserved or affected by morphisms from one thing into another. Starting from a small number of axioms and principles, much like set theory or the lambda calculus, category theory is able to encompass a remarkable amount (arguably all) of mathematics.

It would be impossible to do justice to a description of category theory here, but several excellent books and tutorials are available. The motivated reader might consider any of these, to name just a few:

- Category Theory for Computing Science, M. Barr and C. Wells [1]
- Conceptual Mathematics: a First Introduction to Categories, F. William Lawvere [6]
- Categories for the Working Mathematician, Saunders MacLane [7]

- Basic Category Theory for Computer Scientists, Benjamin Pierce [11]
- Practical Foundations of Mathematics, Paul Taylor [20]

Having said that, we still must present a few simple concepts from category theory, and from Specware's use of it, to make the text in subsequent sections meaningful.

Categories

Categories are composed of objects connected by arrows. There are a few simple constraints we can ignore at this level of discussion, except to note that arrows compose: given an arrow from A to B and another from B to C, there must be[2] the composed arrow from A to C. Also note that, in general, two objects in a category might be connected by any number of arrows, from none to an infinite number, and any number of distinct arrows may loop from an object back to itself, except that a looping identity arrow is always present for each object.

Diagrams

It is reasonable to think of a diagram as a multi-graph with loops in which each node and arc is labeled by an object or arrow, respectively, from some target category. Distinct nodes are allowed to be labeled by the same object, and distinct arcs between the same two nodes (or looping on the same node) are allowed to be labeled by the same arrow.

One important property of diagrams is that they may commute. What this means is that for any two distinct paths in some diagram from node A to node B, the compositions of the arrows labelling the arcs of either path each produce essentially the same arrow. In general, this need not be true for arbitrary diagrams. When we know it is true for all the diagrams under consideration we can exploit this information.

Colimits

For many categories, there is an operation called *colimit* that can be applied to a diagram to glue together all the objects in that diagram, creating a new object, where the precise nature of the gluing is given

[2] perhaps platonically

by the arrows in the diagram. Essentially, it is a means to push the external structure captured by a diagram down into a single object.[3]

The technical definition is that a colimit of a diagram is the apex of the universal cocone over the diagram, where a cocone is a collection of arrows, one for each node in the diagram, all converging on the same target and commuting with each other and the original arrows in the diagram. More informally, a cocone is a way to compatibly combine all the information from a diagram into one object, and a colimit is the object created by a minimal cocone — one that does the least amount of structure sharing to accomplish this goal. Not all categories have colimits for their diagrams, but the ones used by Specware do. Moreover, Specware uses a highly efficient (essentially linear time) algorithm to compute the colimit of diagrams in the base category of specs and spec-morphisms.

To help train your intuitions, note that in the degenerate case where a diagram consists of just two unconnected nodes, each labeled by the same spec S, the colimit spec for that diagram will consist of two distinct and non-interacting copies of S — no arrows means that nothing from either copy was equated with anything in the other.

Also consider the diagram of the three upper left boxes shown in figure 1 below, where *Triv* is the trivial spec containing exactly one sort E, with no operations on it. Assume that $Triv \rightarrow X$ is $\{E \mapsto A\}$ and $Triv \rightarrow Y$ is $\{E \mapsto I\}$.

The colimit spec for this diagram is shown at the lower right, and is similar to the one for $S + S$ above, with almost distinct and non-interacting copies of specs *Triv*, X and Y, except that the sorts E, A and I will be equated in the colimit, thus allowing operations from spec X to interact with those from spec Y via that sort.

5 Basic Categories

The original Specware already contains several categories. In the primary category, objects are algebraic specifications, and arrows are spec-morphisms.

[3] To be precise, the colimit object is already present in the eternal, unchanging category and the colimit merely selects it, but for our purposes here it's better to think of the operation as revealing something new we have learned about the category.

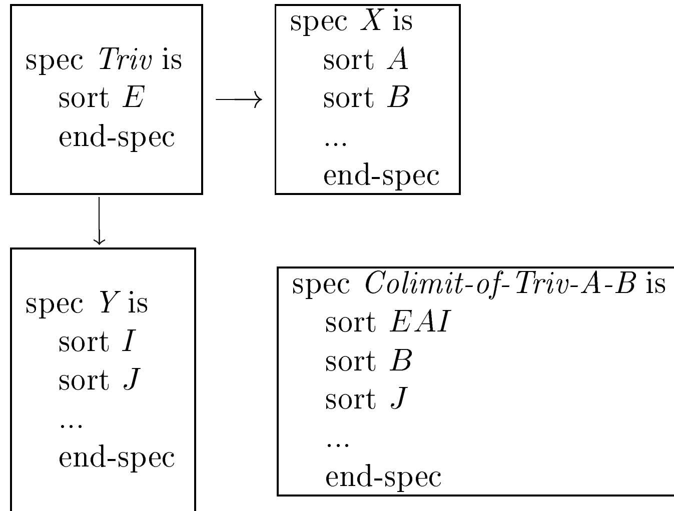

Figure 1: Colimit of a Diagram of Specifications

Other categories used by Specware build on the primary category of specs and spec-morphisms, using arrows or diagrams from one level of category as components to build objects and arrows at new levels. In particular, the category of specs and interpretations, described below, is built this way,

A few additional categories in Specware use variants of the primary category to describe modules in target languages such as Lisp, C, C++, etc.

5.1 Category of Specs and Spec-Morphisms

The primary category in Specware is that of *specs* (algebraic specifications) and *spec-morphisms* (truth-preserving mappings of one spec into another).

Algebraic Specification

An algebraic specification is a collection of named sorts, named typed operations, a set of axioms defining or otherwise constraining those sorts and operations, and theorems derivable from those axioms. Figure 2 shows a typical specifications in Slang for a spec named `CONTAINER` created de-novo, while figure 3 shows the definition of a spec named `PROTO-SEQ` via the composition of prior specs. Figure 4 shows part of a typical specification in MetaSlang for a spec named `SplayTree`.

```
spec CONTAINER is
 sorts E, C
 op empty     : C
 op singleton : E    -> C
 op join      : C, C -> C
 constructors {empty, singleton, join} construct C

 op empty?     : C    -> Boolean
 definition of empty? is
  axiom (empty? empty)
  axiom (not (empty? (singleton x)))
  axiom (iff (empty? (join U V))
             (and (empty? U) (empty? V)))
 end-definition

 op nonempty?  : C    -> Boolean
 definition of nonempty? is
  axiom (iff (nonempty? C) (not (empty? C)))
 end-definition

 op in         : E, C -> Boolean
 definition of in is
  axiom (not (in x empty))
  axiom (iff (in x (singleton y)) (equal x y))
  axiom (iff (in x (join U V)) (or (in x U) (in x V)))
 end-definition

 op insert    : E, C -> C
 definition of insert is
  axiom (equal (insert x C) (join (singleton x) C))
 end-definition

 constructors {empty, insert} construct C

 end-spec
```

Figure 2: Typical Basic Specification in Slang

```
spec PROTO-SEQ is
 translate
   colimit of
     diagram
       %% Monoid = associative + unit
       nodes MONOID, CONTAINER, MONOID-SIG
       arcs MONOID-SIG -> MONOID : {}
          , MONOID-SIG -> CONTAINER : {E     -> C,
                                      binop -> join,
                                      unit  -> empty}
     end-diagram
   by {C -> Seq, empty -> empty-seq, join -> concat}
```

Figure 3: Typical Composite Specification in Slang

```
spec SplayTree =
 sort splay(a) = | SplayObj {value : a,
                             right : splay(a),
                             left  : splay(a)
                            }
                 | SplayNil
  sort ans_t(a) = | No | Eq a | Lt a | Gt a
 ...
 op splay : fa(a) (a -> General.Order) * splay(a)
                  ->
                 General.Order * splay(a)
 def splay (compf, root) =
  case adj compf root
  of (No,_,_)   -> (GREATER,SplayNil)
   | (Eq v,l,r) -> (EQUAL,  SplayObj{value = v,left = l,right = r})
   | (Lt v,l,r) -> (LESS,   SplayObj{value = v,left = l,right = r})
   | (Gt v,l,r) -> (GREATER,SplayObj{value = v,left = l,right = r})
 ...
end-spec
```

Figure 4: Typical Basic Specification in MetaSlang

The Slang language in the original Specware supports sum, product, arrow, subsort, and quotient sorts. The MetaSlang language in newer versions of Specware adds support for polymorphic types and a richer syntax for defining sorts and operations. Both languages support higher-order axioms, i.e., quantification over functions and predicates. In general, these are fairly expressive languages, although the lack of partial functions and the lack of stateful operations is noticeable, and those issues are now active topics at Kestrel.

A theorem prover can be invoked to automatically attempt to verify the theorems from the given axioms when a spec is created. We currently have interfaces to Snark [16] and Gandalf [4], and others can readily be added.

Spec-Morphism

A signature morphism is an arrow that maps each symbol from the source spec into some symbol in the target spec, such that the types of operations are mapped consistently. A spec-morphism is a signature morphism with the added requirement that the symbol mapping translates each axiom of the source theory into a conjecture that can be proved in the target theory. A theorem prover can be invoked to automatically attempt this verification when a morphism is created.

Somewhat remarkably, spec-morphisms can capture most of the notions of importation, parameterization and refinement, and colimits can be used to create composite specifications from diagrams of smaller specifications, instantiate parameters and perform refinements.

5.2 Category of Specs and Interpretations

Spec-morphisms provide a rather literal and limited mapping of one spec into another, since they map symbols only to symbols. This simplicity makes much of the fundamental code in Specware simple and efficient, but in practice, one often wishes to map symbols to expressions.

Rather than complicate the innermost code for Specware, we instead use the categorical tools to construct a new category in which the objects are still just specs, but the arrows, called interpretations, allow symbols to map to expressions.

A *definitional extension*, depicted as $Source \xrightarrow{d} Target$, is a spec-morphism in which any new symbols in the target spec have definitions in terms of the imported source spec. In other words, a definitional extension merely provides names for sorts and operations already implicitly present in the theory. The models of the source and target of a definitional extension will be isomorphic.

An interpretation, depicted as $Source \Longrightarrow Target$, is built from two spec-morphisms, each of which points into a common mediating spec, as follows:

$Source \rightarrow Mediator \xleftarrow{d} Target$

The target-to-mediator morphism must be a definitional extension. The mediator spec provides explicit names for sorts and operations that were implicitly present in the target theory [4], such that every symbol from the source has an appropriate target in the mediator. The net effect is an interpretation arrow that can be viewed as mapping source symbols to target expressions (as opposed to just target symbols).

5.3 Other Categories

In addition to the categories above, the original Specware has categories for language-specific specs (e.g. for Lisp, C, or C++) and morphisms on them which essentially describe a module import relation.

It also has categories whose objects are interpretation-schema (i.e., the objects here are arrows from the category of specs and interpretations), and whose arrows are covariant mappings of those.

Using just the tools above, it was possible to generate code from some specifications, but various problems impelled us to develop two additional categories: one for parameterized-specifications with their refinements, and another for diagrams with diagram-morphisms.

6 Parameterized Specifications

One problem with simple specifications and morphisms is that they only support covariant refinement. For example, given one spec for partial orders

[4] Recall that specs are the finite presentations of theories containing an infinite number of implicit terms and theorems.

and another for sorting, plus a morphism from the former to the latter, it is possible to attach an interpretation restricting sorting to quicksort, and another restricting partial orders to total orders, but this is typically not what is desired. In a sense, the target of a covariant refinement must anticipate the source, hence in general, a restriction to covariant refinement leads to ad hoc solutions whose components cannot be reused.

6.1 Illustration of problem

The problem can be illustrated by the following example, where the goal is to refine an abstract theory for sets of integers into an implementable theory for arrays of bitvectors.

In what follows, the colimit of $\mathit{Set} \leftarrow \mathit{Triv} \rightarrow \mathit{Int}$ is described as *Set(Int)*, and similarly, the colimit of $\mathit{Array} \leftarrow \mathit{Triv} \rightarrow BV$ is *Array(BV)*.

The covariant construction of an interpretation $\mathit{Set(Int)} \Rightarrow \mathit{Array}(BV)$ then needs three vertical interpretations: $\mathit{Set} \Rightarrow \mathit{Array}$, $\mathit{Triv} \Rightarrow \mathit{Triv}$, and $\mathit{Int} \Rightarrow BV$, as follows:

$$\begin{array}{ccccc}
Set & \longleftarrow & Triv & \longrightarrow & Int \\
\Downarrow & & \Downarrow & & \Downarrow \\
Array & \longleftarrow & Triv & \longrightarrow & BV
\end{array}$$

This can be expanded to a diagram of the underlying spec-morphisms as follows:

$$\begin{array}{ccccc}
\mathit{Set} & \longleftarrow & \mathit{Triv} & \longrightarrow & \mathit{Int} \\
\downarrow & & \downarrow & & \downarrow \\
\mathit{Set\text{-}as\text{-}Array} & \longleftarrow & \mathit{Triv\text{-}as\text{-}Triv} & \longrightarrow & \mathit{Int\text{-}as\text{-}BV} \\
\uparrow d & & \uparrow d & & \uparrow d \\
\mathit{Array} & \longleftarrow & \mathit{Triv} & \longrightarrow & BV
\end{array}$$

If the three mediator specs are connected as shown and the diagram commutes, we can compute the colimit of the three interpretations to produce the desired interpretation $\mathit{Set(Int)} \Rightarrow \mathit{Array}(BV)$

The details of the interpretation $\mathit{Set} \Rightarrow \mathit{Array}$ are irrelevant.

The interpretation $\mathit{Int} \Rightarrow BV$ proceeds through *Int-as-BV*, which imports BV and adds the sort *Integer-as-Bitvector* plus operations that define

integers using bit-vectors, so as to provide suitable target symbols for the spec-morphism *Int* $\rightarrow$ *Int-as-BV*.

Now however, any attempt to construct *Triv* $\Rightarrow$ *Triv* will fail.

To see why, first consider the box of arrows at the upper right — it defines two paths from the source spec *Triv* to to the spec *Int-as-BV*. The upper path maps sort *E* from *Triv* to sort *Integer* in *Int*, which is then mapped to sort *Integer-as-BitVector* in *Int-as-BV*. Hence the composite arrow maps sort *E* from *Triv* to sort *Integer-as-BitVector* in *Int-as-BV*. Since the top and bottom paths must commute, this means that if sort *E* from *Triv* maps to sort *E1* in *Triv-as-Triv*, then *E1* in turn must map to sort *Integer-as-BitVector* in *Int-as-BV*.

But by a similar argument using the box of arrows at the lower right, we see that if sort *E* in the target *Triv* maps to *E2* in *Triv-as-Triv*, then *E2* must map to sort *BitVector* in *Int-as-BV*.

Since *E1* and *E2* map to different targets in *Int-as-BV*, they must be different sorts in *Triv-as-Triv*.

But *E1* and *E2* must also map to sorts in *Set-as-Array*.

Then for everything to combine properly in the final result, the structure of the mediating spec *Int-as-BV* will need to be compatible in ad hoc ways with the structure of the mediating spec *Set-as-Array*, destroying modularity and leading to excessive propagation of changes.

6.2 Solution

The solution starts by noticing that the morphism *Triv* $\rightarrow$ *Set* is a parametric specification. Essentially this means that *Set* does not add any axioms within the subtheory it imports from *Triv*, or alternatively, that any model for *Triv* can be expanded into a model for *Set*. Hence for any arrow *Triv* $\rightarrow X$, the colimit of the diagram *Set* $\leftarrow$ *Triv* $\rightarrow X$ cannot introduce inconsistency. This is a highly desirable property.

Not all morphisms are parametric. For example, consider the morphism from spec *BinRel*, containing a sort with one (otherwise unspecified) binary relation R, into spec *Reflexive*, which adds the axiom that $R(x, x)$ holds for all x. Then consider the spec *Irreflexive*, which has one sort and a binary relation R with the axiom that $\neg R(x, x)$ holds for all x. Then the colimit of the diagram *Reflexive* $\leftarrow$ *BinRel* $\rightarrow$ *Irreflexive* would be inconsistent.

Starting from such insights, we developed a new category whose objects are parametric specifications (i.e., spec-morphisms with the property above),

called pspecs for brevity, and whose arrows are ***contravariant*** refinements of pspecs. Pspecs will be indicated in diagrams by $\xrightarrow{p}$.

6.2.1 Contravariance

Contravariant refinement of pspecs is directly analogous to contravariant refinement of functions. To implement $F : A \to B$ in terms of $G : C \to D$ it suffices to show that A is a subset of C, that D is a subset of B, and that for x in A, *DtoC(G(AtoB(x)))* is the desired value for *F(x)*. In effect, we follow arrows around three sides of a square to implement the fourth.

Similarly, to refine $BodyA \xleftarrow{p} ParmA$ into $BodyB \xleftarrow{p} ParmB$ we want $BodyA \Rightarrow BodyB$, but (contravariantly) $ParmB \Rightarrow ParmA$.

6.2.2 Example

Consider an abstract specification for $Sort \xleftarrow{p} TotalOrder$. Then note that Common Lisp has a routine called **sort** which accepts a comparison function as one of its arguments. We can specify the lisp sort routine as $LispSort \xleftarrow{p} LispRel$, where *LispRel* describes a binary relation. In lisp, even with traditional strong type-checking added, there is nothing to prevent us from calling the sort function with a relation that returns arbitrary boolean values, in which case we may get nonsense. However, we also know that if the supplied relation defines a total order then the sort function will return an ordered list. The latter knowledge justifies adding a library refinement that refines $Sort \xleftarrow{p} TotalOrder$ into $LispSort \xleftarrow{p} LispRel$. We will never generate the nonsense cases because the instantiated spec *Sort(RandomRel)* would require an impossible morphism $TotalOrder \to RandomRel$. Hence we can guarantee properties at the problem specification level, and refine those abstract specifications to primitive routines, in a manner that preserves the guarantees, even though using the target system directly could lead to nonsense.

6.2.3 Category of Pspecs

The actual details for the category of pspecs and their refinements are somewhat complex, but are presented clearly in *Refinement of Parameterized Algebraic Specifications* by Yellamraju V. Srinivas [19]. This paper discusses

the alternative semantics available for pspecs, and provides proofs of several desired properties. Among other things, it provides semantics for the following constructions:

- Instantiation of Pspecs

  For example, given *Set* $\xleftarrow{p}$ *Triv* and the morphism *Triv*$\rightarrow$*Int*, generate the instantiated spec *Set(Int)* using the colimit operation.

- Contravariant Refinement of Pspecs

  For example, refine *Sort* $\xleftarrow{p}$ *TotalOrder* into *Array* $\xleftarrow{p}$ *BinRel* using a refinement that contains an interpretation from *BinRel* into *TotalOrder* and an interpretation from *Sort* into *Array(TotalOrder)*.

- Vertical Composition of Contravariant Refinements

  For example, compose a refinement of *Bag* $\xleftarrow{p}$ *TotalOrder* into *Seq* $\xleftarrow{p}$ *PartialOrder* with a refinement of *Seq* $\xleftarrow{p}$ *PartialOrder* into *Array* $\xleftarrow{p}$ *BinRel*, yielding a refinement of *Bag* $\xleftarrow{p}$ *TotalOrder* into *Array* $\xleftarrow{p}$ *BinRel*.

- Horizontal Composition of Pspecs

  For example, compose a refinement of *Set* $\xleftarrow{p}$ *Triv* with *Array* $\xleftarrow{p}$ *Triv* to get *Set(Array)* $\xleftarrow{p}$ *Triv*.

7 Hereditary Diagrams

Hereditary diagrams formalize the notion of designs as diagrams, and make explicit the notion of diagrams of (diagrams of ...) diagrams, defining their properties, the way they should be implemented, and how to compute colimits of them. In the process, this new technology bridges a significant representational discrepancy between the way humans think about diagrams of diagrams and the way computers can effectively implement them.

Moreover, a pleasant discovery is that the notion of parameterized specifications can be captured as specific instances of this more general notion.

The effect of these additions has been to greatly expand the expressiveness of Specware, and to make it possible to codify at any desired level of abstraction software composition, transformation, refinement, evolution, etc.

in term of hereditary diagrams, viz. diagrams of diagrams of ..., with this recursion ultimately grounding in some base category such as that for specs and spec-morphisms. The user is able to access and manipulate these increasingly more abstract concepts using just one set of recursive mechanisms, which is a tremendously liberating new paradigm.

7.1 Diagrams as Designs

Diagrams provide a good mathematical structure for capturing the informal notion of designs. They can be used to describe the *horizontal* composition of a complex design, the *vertical* refinement of a design from one level of abstraction to another, *parameterization* of a design, and *designs of designs*.

Horizontal Composition

We saw small examples of horizontal composition above, in the creation of specifications such as Set(Int) using the diagram $Set \leftarrow Triv \rightarrow Int$. But the technology scales readily to indefinitely larger and more complex objects, arrows, and graphs. For example, the diagram in figure 5 expresses that the components for avionics and on-board communications obey the same on-board network protocol, and that the on-board communications and air-traffic control system each obey the same broadcast protocols.

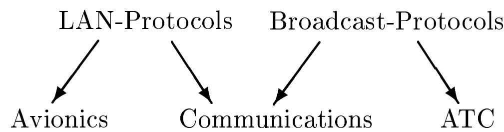

Figure 5: Diagram composing complex specifications

The colimit of this diagram would provide a specification in which one could express interactions between air-traffic control and the avionics software, with rigorous adherence to the broadcast protocols and to behavioral restrictions imposed by the on-board network.

For another example, which we'll refer to later, suppose we want to formally specify a certain kind of operating system (OS). Assuming the spec describing the OS doesn't depend on the particular details of the virtual memory (VM), we can abstract out as a parameter the spec describing the VM. Likewise, a particular VM_0 might not depend on

the details of the paging policy (PP), so we could abstract a second parameterization of VM_0 by PP.

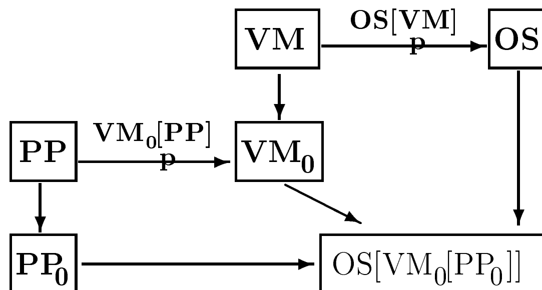

Figure 6: Diagram composing parameterized specifications

Vertical Refinement

We also saw small examples of vertical refinement above, for example the implementation of sets as arrays: *Set* ⇒ *Array*. Traditional implementation techniques such as compilers tend to go in one or two fixed logical steps from user-provided specifications to machine-executable implementation. Making the refinement steps explicit and composable provides a whole new realm of flexibility for expressing the final implementation.

The level of detail provided by Specware also allows for alternative refinement of the same abstract data structure in alternative contexts. E.g., one set might be implemented as a stack, another as an array, and a third as a bitvector.

Future work might even allow the same object to be implemented in different ways in different parts of the system, with automatically generated transitions at the phase boundaries.

Orthogonal Development Paths

The horizontal composition arrows and vertical refinement arrows are expressed using the same structure, and have the important property that they *commute*. What this means is that developers are not required to follow a fixed development path — they can do some horizontal composition of components, then some vertical refinements of all or part of those, then more composition, more refinement, etc.

7.2 Lazy Colimits

Recently we have developed technology to make diagrams more "first-class" within Specware. In particular, we shifted from doing colimits of diagrams eagerly, before various other operations are performed, to doing colimits lazily, as late as as possible in the development process.

7.3 Problems with old Approach

Older versions of Specware use colimits *eagerly*: the developer makes a diagram of specs, takes the colimit, and then proceeds to do refinement etc. on the colimit spec. This collapses and hides information inside the colimit spec. In fact, when it comes time to refine such a spec, machinery inside Specware recovers the original diagram as a cover for the colimit spec, i.e. a structured decomposition into pieces that collectively describe the composite object. From there, the user can refine the component specs and Specware machinery will automatically combine the results using the chosen cover.

There are several problems with this approach, all stemming from the fact that structure which was explicit, visible, and modifiable by users has been made implicit, invisible, and restricted to a set of operations built into Specware.

In the example above for a parameterized operating system, the colimit spec $OS[VM_0[PP_0]]$ has been "cooked" to glue together all of the structure that was explicit in the original diagram. Except by remembering the origin of the colimit spec, there is no simple way to recover that diagram structure.

7.4 Promise of First Class Diagrams

Our design for heritary diagrams does colimits *lazily*: it keeps the original structure visible as long as possible, performing the colimits at the last possible moment.

This laziness implies that diagrams acquire a more prominent status within Specware. Instead of being primarily the scaffolding used by Specware to manipulate specifications, they become explicit objects in their own right. The user can view them and manipulate them directly, instead of being limited by reliance on ad hoc mechanisms.

Moreover, because the diagrams are more explicit and visible, new kinds of structure can be expressed, especially with respect to parameters.

7.5 Category of Diagrams

For diagrams truly to be "first-class" objects in Specware, we need be able to build categories of diagrams, hence we need certain precise definitions for diagrams and for the arrows that connect them.

A *diagram* is just a functor $D : S \Rightarrow C$, where S is a small[5] category, the *shape* of D, and C is some arbitrary base or target category. For the purposes of Specware, C will tend to be a category such as that for specs and spec-morphisms, or a category of diagrams.

Technically, a *diagram morphism* $f : A \rightarrow B$ is a pair $f =< f_0, f' >$ of a functor $f_0 : A_0 \rightarrow B_0$ and a natural transformation $f' : A \rightarrow B \circ f'$.

$$
\begin{array}{ccc}
A_0 & \xrightarrow{A} & \\
\downarrow f_0 & f' \downarrow & C \\
B_0 & \xrightarrow{B} &
\end{array}
\qquad (1)
$$

In other words, there is a *shape morphism* f_0 that maps the shape of one diagram into the shape of another, and a *relabeling morphism* f' that maps the labeling of A_0 done by the first diagram into the labeling for A_0 that would be implied by traveling first through the arrow f_0 on the way to the target category. This allows us to map one diagram into another where the shape has been changed and the nodes and arcs have been relabeled.

7.6 OS Example Revisited

Let us reconsider the diagram above for the parameterized operating system, but now exploiting the new paradigm of diagrams of diagrams. Instead of treating the OS diagram as one flat diagram and using colimit eagerly to collapse the structure, we will encapsulate portions of the diagram into subdiagrams. For example, figure 7 shows the exposed parameterization $VM \xrightarrow{p} OS$ being applied to the encapsulated parameterization $PP \xrightarrow{p} VM$.

[5] "Small" is a technical term simply meaning that its possible to put all the arrows in some set. A "small" category can be infinite.

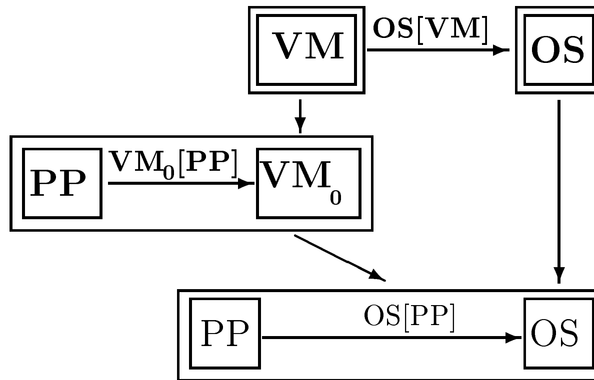

Figure 7: Parameterized specs via heritary diagrams

Combining those two subdiagrams as shown yields a third parameterized diagram for *PP* $\xrightarrow{p}$ *OS*. We have thus captured much of the mechanism previously available for parameterized specs, but using just operations on diagrams.

7.7 Novel Kinds of Parameterization

With diagrams of diagrams, though, we are not limited to the old kinds of manipulations on parameters. For example, consider thes diagram in figure 8, expressing *X* $\xrightarrow{p}$ *List[Array[X]]*.

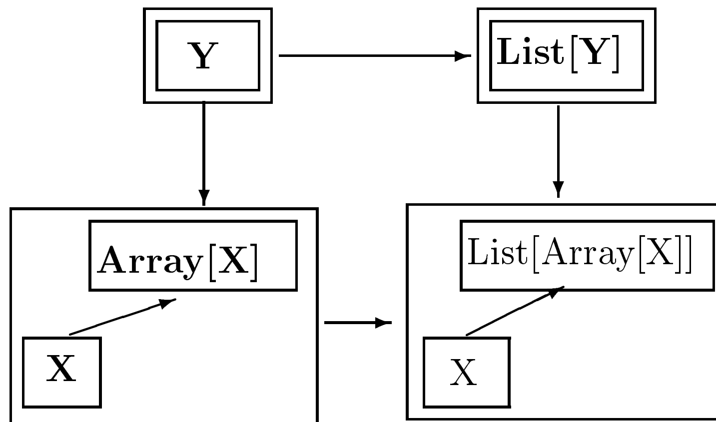

Figure 8: List of Array of X

By simply shifting our focus, we can also extract the parameter *List[Y]* $\xrightarrow{p}$ $List[Array[X]]$, and use that parameterization to construct *List[Y]* $\xrightarrow{p}$ $Set[List[Array[X]]]$ as follows:

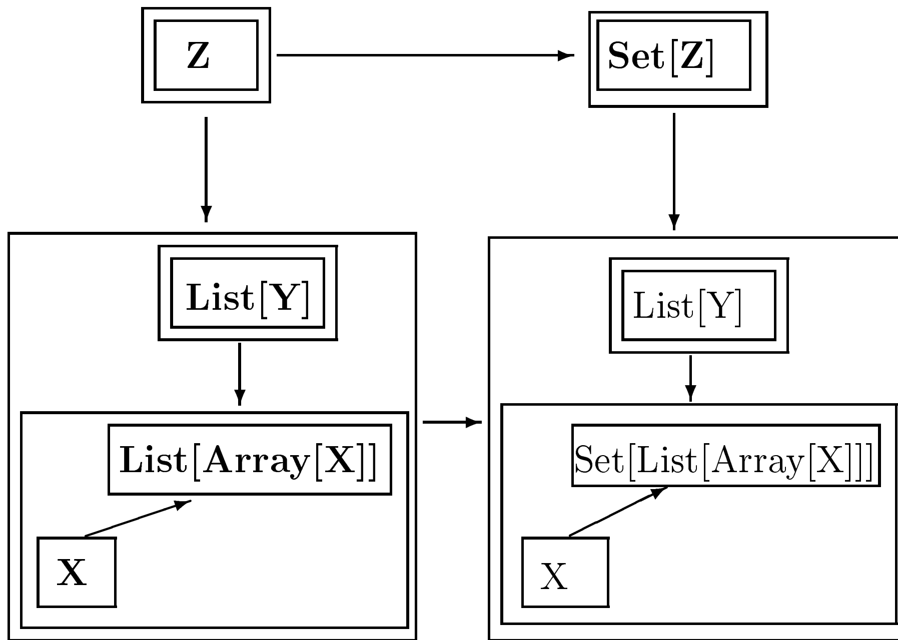

Figure 9: Novel parameterizations

This is something new. Instead of parameterizing merely on the innermost element type, as traditional languages would allow, we have constructed a parameterization of the "middle" structure in a construct, with upper and lower structures fixed (or separately parameterized). And it was done lazily. Long after the diagrams are constructed we can go back and simply reexamine them to exploit new kinds of parameterization — no one had to anticipate our future needs.

This kind of lazy discovery of parameters is now used extensively by $Designware^{TM}$ [6], and forms the basis of the *ladder diagrams* discussed in various related publications ([2] [13] [14] [15]).

The diagram in figure 10 (from [15]) shows how a diagram pushout may be used in practice.

[6]Designware is a trademark of Kestrel Development Corporation

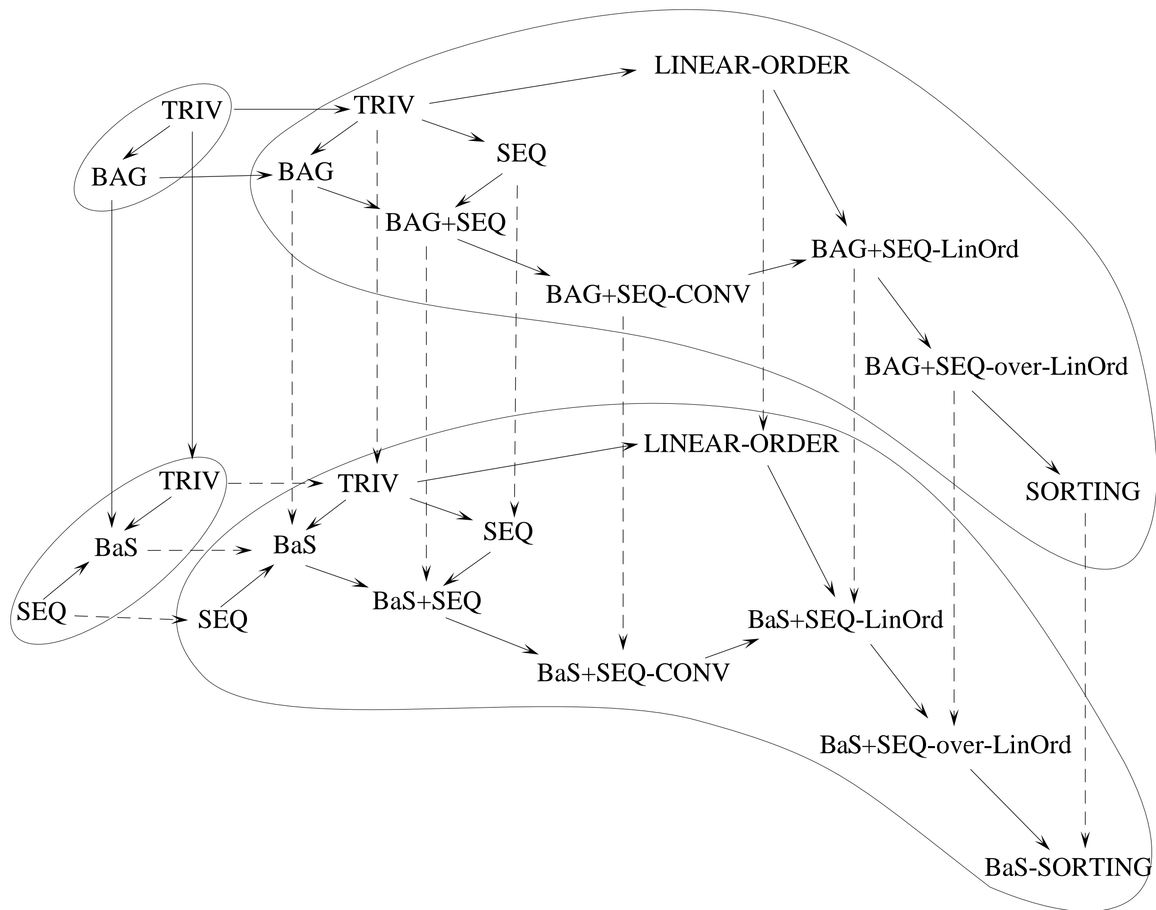

Figure 10: Refining Bags to Seqs in Sorting

7.8 Category of Diagrams

All of the above technology sounds nice, but how can one implement it? As it happens, naive approaches will not work.

The correct computation of colimits in the category of diagrams involves what are known as left Kan extensions, which are a somewhat advanced concept from category theory. They are related to the notion of adjunction, and first arose in the context of describing logical quantifiers. A full explanation of the theory here is beyond the scope of this report — the motivated reader is referred to the category texts recommended above.

The algorithm we developed for computing colimits in the category of

diagrams is efficient and scales well to large diagrams. Kestrel has a patent pending which describes this algorithm, and at some point it also will be described in a forthcoming paper [10].

An important aspect of this algorithm is that it recursively invokes the mechanism for computing colimits of the component diagrams. This leads us to the next big idea: indefinite levels of diagrams of (diagrams of ...) diagrams.

7.9 Category of Designs – Diagrams of Diagrams

Given a specification (or any other kind of object, for that matter) we can create a one-node diagram that is essentially equivalent to it. Now we are in the category of diagrams, and by applying our new tools, we can create new diagrams at successive levels of abstraction.

There is an interesting issue of efficiency. Humans tend to view the diagrams as nested structures, where each node in a diagram at one level "explodes out" to a diagram at the next level down. This is an accurate description of the information, but implementing that model in the computer would lead to exponentially long computations for colimits — the system would be unusable.

To avoid this problem, our technology uses an alternative representation employing *opfibrations*, which essentially capture the idea of a category acting as an index for another category: subcategories in the main category are mapped to nodes in the indexing category, and arrows between nodes in different subcategories are mapped to arrows in the indexing category.

Our representation for diagrams of diagrams uses towers of *opfibrations* as a kind of multi-level indexing scheme, to avoid the exponential growth that an explicit construction would involve.

$$\begin{array}{ccccccc} A_0 & \longleftarrow & A_1 & \longleftarrow \cdots \longleftarrow & A_n & & \\ \downarrow {\scriptstyle f_0} & & \downarrow {\scriptstyle f_1} & & \downarrow {\scriptstyle f_n} & \searrow & C \\ B_0 & \longleftarrow & B_1 & \longleftarrow \cdots \longleftarrow & B_n & \nearrow & \end{array} \tag{2}$$

The category of designs thus consists of

- objects: finite towers of opfibrations, with a diagram at the top $A_0 \longleftarrow A_1 \longleftarrow \cdots \longleftarrow A_{n-1} \longleftarrow A_n \longrightarrow C$
- morphisms: ladders of opcartesian functors, with a diagram morphism at the top.

In essence, we have developed technology that allows us to manipulate these indexing schemes, without paying the price of actually expanding the structure they describe. That operation needs to be done only once at the very end of the software development process, and only for the final structures.

References

[1] Barr, M. and Wells, C. *Category Theory for Computing Science*, Prentice Hall, 1990.

[2] Blaine, L., Gilham, L., Liu, J., Smith, D., , and Westfold, S. Planware – domain-specific synthesis of high-performance schedulers. In *Proceedings of the Thirteenth Automated Software Engineering Conference* (October 1998), IEEE Computer Society Press, pp. 270–280.

[3] Boehm, B. *Software Engineering Economics*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[4] Tammet, T. Reference Manual GANDALF version c-2.0b Department of Computing Science, University of Göteborg / Chalmers Univ. of Technology, S-412 96 Göteborg, Sweden, Computing Centre, Tallinn University of Technology, Raja 15, Tallinn, Estonia, January 2000.

[5] Huang, X. A Comparison Between Standard and Formal Mathematical Software Development. Master of Science Thesis, University of Maryland Department of Nuclear Materials and Reliability Engineering, 1999.

[6] Lawvere, F. W. and Schanuel, S. *Conceptual Mathematics: a First Introduction to Categories*, Cambridge University Press, 1997.

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

[8] PAULK, M. C., CURIS, B., CHRISSIS, M. B., AND WEBER, C. V Capability Maturity Model, Version 1.1. In *IEEE Software, Vol. 10, Number 4*, IEEE Computer Society, Los Alamitos, CA., July 1993.

[9] PAVLOVIĆ, D. Semantics of first order parametric specifications. In *Formal Methods '99* (1999), J. Woodcock and J. Wing, Eds., Lecture Notes in Computer Science, Springer Verlag. to appear.

[10] PAVLOVIĆ, D. Compositionality via diagrams in software design. In progress.

[11] PIERCE, B. *Basic Category Theory for Computer Scientists*, MIT Press, 1991.

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

[13] SMITH, D. R., PARRA, E., AND WESTFOLD, S. J. Generic Tools for Transportation Planning and Scheduling. Final Technical Report *RL-TR-95-143*, Rome Laboratory, Air Force Materiel Command, Griffiss Air Force Base, N.Y., August, 1995.

[14] SMITH, D. R. Toward a classification approach to design. In *Proceedings of the Fifth International Conference on Algebraic Methodology and Software Technology, AMAST'96* (1996), vol. LNCS 1101, Springer-Verlag, pp. 62–84.

[15] SMITH, D. R. Mechanizing the development of software. In *Calculational System Design, Proceedings of the NATO Advanced Study Institute*, M. Broy and R. Steinbrueggen, Eds. IOS Press, Amsterdam, 1999, pp. 251–292.

[16] STICKEL, M. E., WALDINGER, R. J., CHAUDHRI, V. K. A Guide to SNARK. Technical Note Unassigned, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025, May 2000. Available at http://www.ai.sri.com/pubs/technotes/aic-tn-2000:Unassigned/

[17] SRINIVAS, Y. V., AND JÜLLIG, R. Specware: Formal support for composing software. In *Proceedings of the Conference on Mathematics of Program Construction*, B. Moeller, Ed. LNCS 947, Springer-Verlag, Berlin, 1995, pp. 399–422.

[18] SRINIVAS, Y. V., AND MCDONALD, J. The Architecture of $SPECWARE^{TM}$, a Formal Software Development System. Technical report KES.U.96.7, Kestrel Institute, Palo Alto, CA., August 1996.

[19] SRINIVAS, Y. V. Refinement of Parameterized Algebraic Specifications. In *IFIP TC2 Working Conference on Algorithmic Languages and Calculi*, Chapman & Hall, Le Bischenberg, France, February, 1997.

[20] TAYLOR, P. *Practical Foundations of Mathematics*, Cambridge University Press, Cambridge, U.K., 1999.