Automated Software Engineering, 8, 7–30, 2001

Industrial Applications of Software Synthesis via Category Theory—Case Studies Using Specware

KEITH WILLIAMSON keith.williamson@boeing.com
MICHAEL HEALY
The Boeing Company, Seattle, Washington, USA

RICHARD BARKER
EC Wise, San Fransisco, CA, USA

Abstract. Over the last two years, we have demonstrated the feasibility of applying category-theoretic methods in specifying, synthesizing, and maintaining industrial strength software systems. We have been using a first-of-its-kind tool for this purpose, Kestrel's Specware™ software development system. In this paper, we describe our experiences and give an industrial perspective on what is needed to make this technology have broader appeal to industry. Our overall impression is that the technology does work for industrial strength applications, but that it needs additional work to make it more usable. We believe this work marks a turning point in the use of mathematically rigorous approaches to industrial strength software development and maintenance.

It is interesting to note that when this technology is applied to software systems whose outputs are designs for airplane parts, the design rationale that is captured is not only software engineering design rationale, but also design rationale from other engineering disciplines (e.g., mechanical, material, manufacturing, electrical, human factors, etc.). This suggests the technology provides an approach to general systems engineering that enables one to structure and reuse engineering knowledge broadly.

Keywords: category theory, software synthesis, specware, systems engineering

1. Introduction

With the advent of intelligent computer aided design systems, companies such as Boeing are entering an era in which core competitive engineering knowledge and design rationale is being encoded in software systems (Williamson and Healy, 2000). The promise of this technology is that this knowledge can be leveraged across many different designs, product families, and even different uses (e.g., manufacturing process planning as well as product design). However, this promise has been hard to achieve. The reasons for this are complex, but a large challenge arises from the attempt to reuse software.

Software is the end artifact of a long and complicated process that goes from requirements, through a process of design, to an implementation, which is built on top of a virtual machine. In this process, many constraints and assumptions (from both the requirements and the virtual machine) come into play, often in subtle ways, affecting design decisions and ultimately the software itself. In looking at software components, or their specifications, it is often difficult to understand what constraints and assumptions led to their particular

formulation. Errors of understanding lead to errors in system implementation, sooner or later. Even if the constraints and assumptions are understood, and they fit the current needs, they may not fit a future need. When that happens, the challenge of understanding the original design rationale makes it difficult to update and maintain software systems.

A fundamental problem in this paradigm of reuse is that what we are trying to reuse is *software*—the end artifact of a long and complicated process. Knowledge sharing and reuse cannot easily and uniformly occur at the software level alone.

Within the field of automated software engineering, there is an approach to software development and maintenance that appears to solve some of these problems. In essence, this paradigm for software development and maintenance is one that allows the capture and structuring of formal requirement specifications, design specifications, implementation software, and the refinement processes that lead from requirements to software. In this approach, the refinement process can guarantee correctness of the synthesized software. By recording, modifying, and then replaying the refinement history, we are able to more easily maintain the software. By capturing, abstracting, and structuring the knowledge in a modular fashion, we are able to more easily reuse this knowledge for other applications. Knowledge reuse occurs at whatever level of abstraction is most appropriate. Sometimes that level is a domain theory that is involved in stating requirements. Sometimes it is a design pattern. Sometimes it is a software component. Often it is a combination of these.

In this paper, we describe our experiences in applying a category theory based approach to software specification, synthesis, and maintenance. In addition, we give an industrial perspective on what is needed to make this technology have broader appeal to industry. We begin with some formal preliminaries and a brief discussion of the software development tool Specware™.

2. Formal preliminaries

The field of category theory (Crole, 1993; MacLane, 1971; Pierce, 1994) provides a foundational theory. This theory was applied to systems theory and systems engineering (Burstall and Goguen, 1980; Goguen, 1970). This theory was embodied in the software development tool Specware™ (Smith, 1999; Srinivas and Jullig, 1995; Waldinger, 1996).

2.1. Category of signatures

A *signature* consists of the following:

1. A set S of sort symbols
2. A triple $O = \langle C, F, P \rangle$ of operators, where:
   - C is a set of sorted constant symbols,
   - F is a set of sorted function symbols, and
   - P is a set of sorted predicate symbols.

A *signature morphism* is a consistent mapping from one signature to another (from sort symbols to sort symbols, and from operator symbols to operator symbols). The category

Sign consists of objects that are signatures and morphisms that are signature morphisms. Composition of two morphisms is the composition of the two mappings.

2.2. Category of specifications

A *specification* consists of:

1. A signature Sig $= \langle S, O \rangle$, and
2. A set Ax of axioms over Sig

Given two specifications $\langle$Sig1, Ax1$\rangle$ and $\langle$Sig2, Ax2$\rangle$, a signature morphism M between Sig1 and Sig2 is a *specification morphism* between the specifications iff:

$$\forall a \in \text{Ax1}, (\text{Ax2} \vdash M(a))$$

That is, every one of the axioms of Ax1, after having been translated by M, can be proved to follow from the axioms in Ax2. This assures that everything that is provable from Ax1 is provable from Ax2 (modulo signature translation). Of course, Ax2 may be a stronger theory. The category **Spec** consists of objects that are specifications and morphisms that are specification morphisms. In general, specifications can be used to represent problem statements, theories, designs, architectures, or programs. Several example specifications are given in the Figures of this paper.

2.3. Diagrams and colimits

A *diagram* in a category C is a collection of vertices and directed edges consistently labeled with objects and arrows (morphisms) of C. A diagram in the category **Spec** can be viewed as expressing structural relationships between collections of specifications.

The *colimit* operation is the fundamental method in Specware^{TM} for combining specifications. The operation takes a diagram of specifications as input and yields a specification, commonly referred to as the colimit of the diagram. See figures 2 and 3 for examples. The colimit specification contains all the elements of the specifications in the diagram, but elements that are linked by arcs in the diagram are identified (unified) in the colimit. Informally, the colimit specification is a shared union of the specifications associated with each node of the original diagram. Shared here means that sorts and operations that are linked by the morphisms of the diagram are identified as a single sort and operation in the colimit specification. In the example in figure 3, the sort *part* of the specification **parts** is linked with both the sort *panel* of the specification **panels** and the sort *part* of the specification **manufactured parts** (the signature morphisms are not depicted in the Figures). In the colimit specification, these sorts are identified (unified) as the sort *panel*. Likewise, other sorts and operations in the **parts** specification are linked with elements in the specifications **panels** and **manufactured panels**, and then unified in the colimit specification.

The colimit operation can be used to compose any specifications. The fact that specifications can be composed via the colimit operation allows us to construct complex specifications

by combining simpler specifications modularly, just as complex systems can be composed from simpler modules.

3. Specware—a software development tool

Specware™ is software development and maintenance environment supporting the specification, design, and semi-automated synthesis of correct-by-construction software. It represents the confluence of capabilities and lessons learned from Kestrel's earlier prototype systems (KIDS (Smith, 1991), REACTO (Wang and Goldberg, 1991), and DTRE (Blaine and Goldberg, 1991)), grounded on a strong mathematical foundation (category theory). The current version of Specware™ is a robust implementation of this foundation. Specware™ supports automation of:

- Component-based specification of programs using a graphical user interface
- Incremental refinement of specifications into correct code in various target programming languages (e.g., currently C++ and LISP)
- Recording and experimenting with design decisions
- Domain-knowledge capture, verification and manipulation
- Design and synthesis of software architectures
- Design and synthesis of algorithm schemas
- Design and synthesis of reactive systems
- Data-type refinement
- Program optimization

The Specware™ system has some of its roots in the formal methods for software development community. Within this community, there are numerous languages that have been used for specifying software systems; e.g., Z (Spivey, 1992), VDM (Bjorner and Jones, 1982), and Larch among many others (Gannon et al., 1994). Of the many formal specification languages, the Vienna Development Method (VDM) is one of the few that tries to formally tie software requirement specifications to their implementations in programming languages. This system is perhaps the closest to Specware™ in that it allows an engineer to specify a software system at multiple levels of refinement. VDM tools allow for the capture and discharging (sometimes manually) of "proof obligations" that arise in the refinement of a specification from one level to another.

Specware™ differs from VDM by having category theory as its fundamental underlying theory. This appears to give several benefits. It allows for a greater level of decomposition, and then composition, of specifications (the use of diagrams and colimits provides a general framework for this). It provides a solid basis (Goguen and Burstall, 1992; Meseguer, 1989) for preserving semantics in all refinement operations—not only within Slang (Specware's specification language), but also across different logics (e.g., from the logic of Slang into the logics underlying target programming languages). It allows for parallel refinement of diagrams (Waldinger et al., 1996), which helps with the scalability of the technology. Parallel refinement allows the independent refinement of component specifications in a diagram (as long as the refinements are consistent with respect to shared objects (Waldinger et al.,

1996)). Multiple categories (e.g., signatures, specifications, interpretations, and shapes) underlie Specware^{TM}. In each category, the notions of composition, diagram, and colimit, have analogous meanings. The value of category theory lies in its general applicability to multiple situations.

4. Stiffened panel layout

We began our evaluation of this technology with an example of a software component that was being considered for inclusion in a component library for structural engineering software (Williamson and Healy, 1997, 2000). The software component solves a structural engineering layout problem of how to space lightening holes in a load-bearing panel. The component was originally part of an application that designs lay-up mandrels, which are tools that are used in the manufacturing process for composite skin panels. At first glance, the software component appears to solve the following one-dimensional panel layout task (this was pulled from a comment in the header of this component). Given a length of a panel, a minimal separation distance between holes in the panel, a minimal separation distance between the end holes and the ends of the panel, and a minimum and maximum width for holes, determine the number of holes, and their width, that can be placed in a panel (see figure 1). This software component solves a specific design task that is part of a broader design task. Prior to the invocation of this function, a structural engineer has determined a minimum spacing necessary to assure structural integrity of the panel.

Upon closer inspection of the software, one realizes that this function actually minimizes the number of holes subject to the constraints specified by the input parameter values. The original set of constraints defines a space of feasible solutions. Given a set of parameter values for the inputs, there may be more than one solution to picking the number of holes and their width so that the constraints are satisfied. So, the software documentation is incomplete. However, going beyond this, one is inclined to ask, "Why did the programmer choose to minimize the number of holes?" Is there an implicit cost function defined over the feasible solutions to the original set of constraints? If so, what is it? Presumably, this is all part of the engineering design rationale that went into coming up with the (not fully stated)

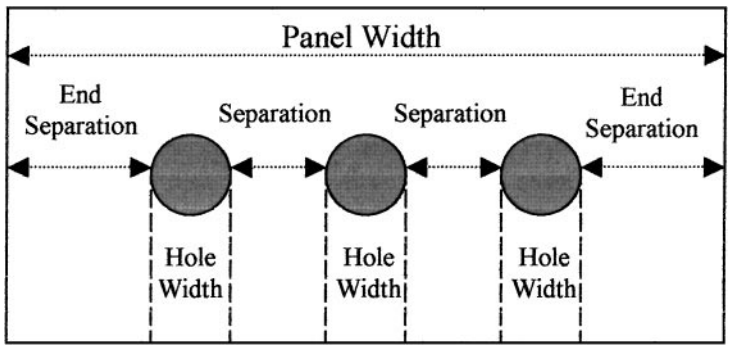

Figure 1. Picture for panel layout problem.

specification for the software component in the first place. If we were to use this component to design a panel that was to fly on an airplane, the panel would be structurally sound, but not necessarily of optimal cost (e.g., not making the best trade-off between manufacturing cost and overall weight of the panel).

Rather than put this incompletely documented software component into a reuse library, we seek to explicate the engineering requirements and design rationale and tie them directly to the software. For this purpose, we used the Specware™ system to first document the structural and manufacturing engineering requirements, then document the engineering design rationale leading to the software component specification, and then generate software that provably implements the specification (which in turn requires documenting software design rationale). In all these cases, the design rationale that was captured was purely functional in nature. Non-functional design rationale (e.g., performance criteria for derived software) is more difficult to express formally, was not captured.

Specware™ allows specifications to be composed in a very modular fashion (using the colimit construction from category theory). In this example, we were able to generate a specification for basic structural parts by taking a colimit of a diagram, which relates specifications for basic physical properties, material properties, and geometry (see figure 2). A specification for stiffened panels was derived by taking the colimit of another diagram, this time relating basic structural parts to panel parts and manufactured parts (see figure 3). This specification was then imported into another, which added manufacturing properties

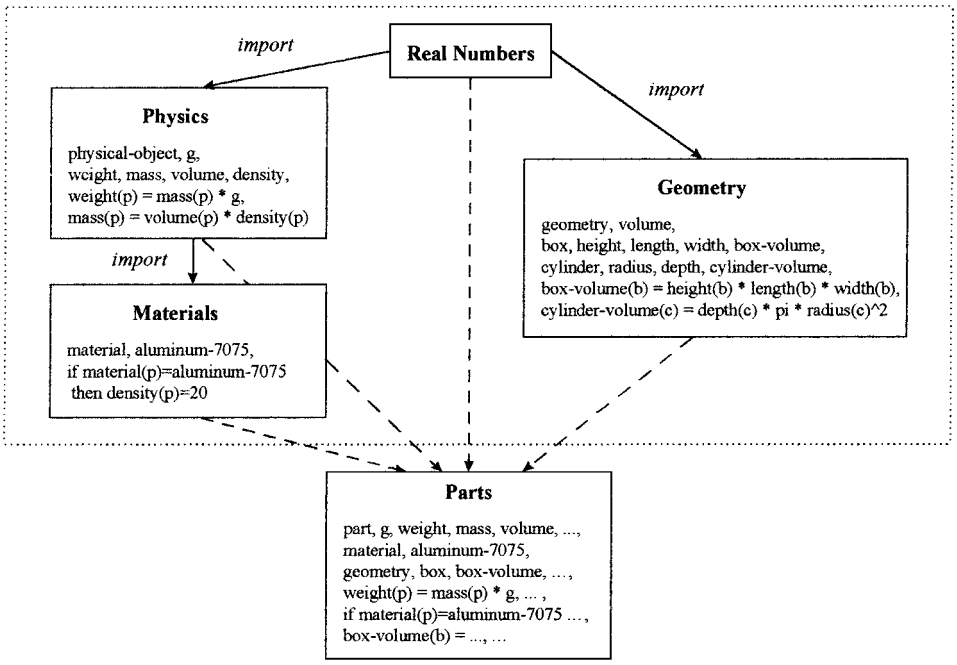

Figure 2. Colimit of a specification diagram.

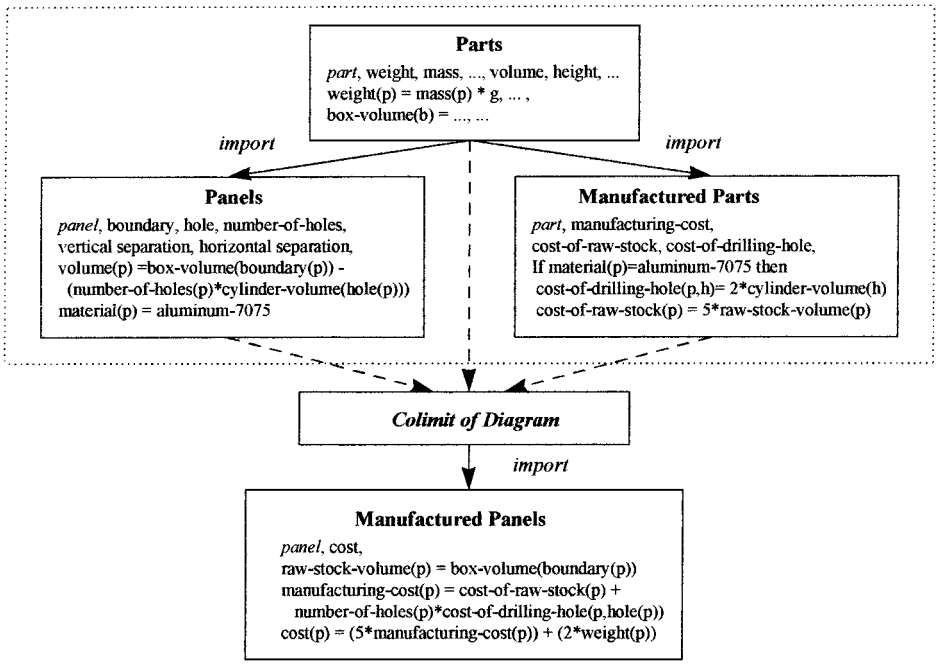

Figure 3. Another colimit and specification morphism.

that are specific to stiffened panels (it is here that we finally state the (originally implicit) cost function). From this specification, and another describing basic optimization problems, we are able to formally state the panel layout problem. This specification was then refined into Lisp software (Williamson and Healy, 2000).

4.1. The challenge of software maintenance

As business processes change, software requirements must change accordingly. Some software changes are straightforward. Other changes are harder to make, and the inherent complexity is not always obvious. In the stiffened panel layout example, suppose there is a change to the material of the panel. If the density of the new material is less than five, then the search algorithm that was used is no longer applicable (Williamson and Healy, 2000). In fact, with this single change to the cost function, it is more cost effective to have no holes in the panel at all!

What is fundamentally missing in the software (the end artifact in the software development process) is the fact that a design decision, that of picking a particular algorithm to solve a class of optimization problems, is reliant on a subtle domain constraint. Indeed, in the original software component, the cost function is nowhere to be seen in the software, nor the documentation that was associated with it. Knowledge sharing and reuse cannot easily and uniformly occur at the software level alone.

If we place requirement specifications, design specifications, and software derivations in a repository, we can reuse them to derive similar engineering software. When requirements change (e.g., in response to a change in manufacturing processes), we are able to change the appropriate specifications, and attempt to propagate those changes through the derivation history. Sometimes the software can be automatically regenerated. Other times, some of the original software design will break down (due to some constraints no longer holding). In this case, we need to go back to the drawing board and come up with a new design to satisfy the requirements. But even in these cases, presumably only some portions of the software will need to be redesigned. We leverage those parts of the software design that we can. In this way, we reuse knowledge at the appropriate level of abstraction, and not solely at the software level.

5. Equipment locator problem

After the successful experience of using Specware^{TM} on the stiffened panel layout component, we decided to see if the technology would scale up to industrial strength applications. There were various criteria that we used to pick this application. The application should:

- Be large enough to test scalability.
- Be a real problem of importance to Boeing.
- Already have requirement documents written.
- Be an engineering application with relatively simple geometry.
- Have requirements that change over time.
- Be in a family of related applications.
- Have overlap with the panel layout problem.
- Be functional in nature (i.e., no real time constraints).

Some of these requirements were chosen in an effort to maximize reuse of knowledge over time and across applications. We have felt that the additional up front costs (associated with rigorously defining requirement specifications and design specifications) can more easily be justified if there is a high probability that those costs can be amortized over a relatively short period of time (i.e., two or three years). Only the last requirement is due to current state of the technology (although work is being in this area).

After some searching, we found the equipment locator problem, which satisfied all of the criteria listed above. This is the problem of determining optimal placements of electronic pieces of equipment (e.g., the flight data recorder, inertial navigation systems, flight computers, etc.) on shelves of racks in commercial airplanes. See figure 4 for a CAD drawing showing three racks, with a piece of equipment placed on the bottom shelf of one of the racks. The purpose of the equipment locator application is to support the equipment installation design process for determining optimal locations for electrical equipment. The application supports the designers in determining optimal locations for equipment on a new airplane model, as well as finding a suitable location for new electrical equipment on an existing airplane model. The application is intended to reduce the process time required

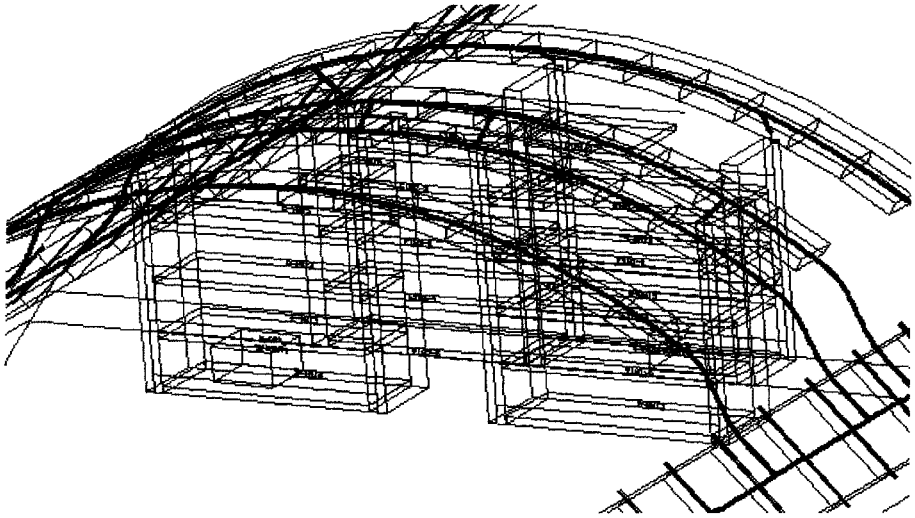

Figure 4. CAD drawing for equipment locator problem.

to determine equipment locations, and also improve the quality of the equipment location designs.

Numerous specifications are needed for stating and solving this problem; e.g.:

- Theory of geometry,
- Global and relative part positioning,
- Major airplane part and zone definitions,
- Operations and properties for pieces of equipment, shelves, and racks,
- Separation and redundancy requirements for equipment,
- An assignment of a piece of equipment to a position on a shelf,
- A layout (a set of assignments),
- Hard constraints on layouts,
- Cost function on layouts,
- Theories of classes of optimization problems,
- Theories of search algorithms for classes of optimization problems,
- The equipment locator problem statement,
- An algorithmic solution to the problem (instantiating a branch and bound algorithm)

All in all, about 7,000 lines of requirement and design specifications are needed to state and solve this problem. The generated Lisp software exceeds 7,000 lines.

5.1. General problem statement

With some simplification, the equipment locator problem has as inputs:

1. A set of shelves
2. A set of equipment
3. A partial layout of equipment to shelves

And produces as output the set of all layouts of equipment to positions on shelves, such that:

1. The partial layout is preserved/extended,
2. All pieces of equipment are placed in some position on some shelf,
3. All hard constraints are satisfied,
4. Layout costs (e.g., wiring distances between pieces of equipment) are minimized.

The hard constraints, which define feasible layouts, are things like:

- Equipment assignments can not overlap in space,
- Equipment must be placed on shelves with appropriate cooling properties,
- Redundant pieces of equipment must be placed on separate cooling systems,
- Critical pieces of equipment have certain restricted regions in space,
- Equipment with low mean time to failure must be easily accessible,
- Voice and flight data recorders must be placed in the front electrical engineering bay,
- Equipment sensitive to electromagnetic interference must be separated (by a certain distance) from other equipment emitting that interference.

The cost function on layouts includes such things as:

- Equipment wiring distances minimized,
- Heavy equipment should be placed as low as possible (for ergonomic considerations),
- Voice and flight data recorders should be placed as far aft as possible.

As an example of formal requirement specifications, figure 5 contains three examples (Specware™ does have a infix notation, but we did not use it). The first constraint states that for any two assignments in a layout, they cannot be overlapping in space. The second constraint states that if two pieces of equipment are redundant with each other (e.g., redundant flight control computers), then the distance between their placements must be greater than the minimal separation distance that is required for that pair of equipment. The third constraint states that redundant pieces of equipment must be placed on shelves that have different types of cooling systems.

5.2. Process for technology use

So how does one go about using this technology for industrial applications? After having learned some of the underlying theory, and then Specware™ (from working on the stiffened panel layout problem), we proceeded to learn the domain of our new application. We had three English requirement documents to work from. These comprised about 20

```
op no-overlapping-assignments : layout -> boolean
axiom (iff (no-overlapping-assignments l)
           (fa (a1:assignment a2:assignment)
               (implies (and (in a1 l) (in a2 l))
                        (implies (not (equal a1 a2)) (not (overlapping a1 a2))))))

op redundant-separated-enough : layout -> boolean
axiom (iff (redundant-separated-enough l)
           (fa (a1:assignment a2:assignment)
               (implies (and (in a1 l) (in a2 l))
                        (implies (redundant (equipment-of a1) (equipment-of a2))
                                 (gt (min-distance (assigned-geometry a1) (assigned-geometry a2))
                                     (redundant-sep (equipment-of a1) (equipment-of a2)))))))

op redundant-separate-cooling : layout -> boolean
axiom (iff (redundant-separate-cooling l)
           (fa (a1:assignment a2:assignment)
               (implies (and (in a1 l) (in a2 l))
                        (implies (redundant (equipment-of a1) (equipment-of a2))
                                 (not (equal (cooling (shelf-of a1)) (cooling (shelf-of a2)))))))))
```

Figure 5. Some example requirement specifications.

pages of writing, drawings, etc. In addition, we had several pieces of supporting material (tables, drawings, etc). Only two discussions with a domain expert were needed, since the requirement documents were fairly clear and complete.

Once understood, we formalized the requirements. Part of this involved figuring out how to best decompose and abstract various portions of the problem domain. We estimate that we captured roughly 98% of the requirements found in the informal material. The remaining 2% of the requirements dealt with interfacing with other software systems (of which we had insufficient information at the time). Next, we went through a manual validation process in which we compared the formal requirements with the informal ones. We wrote a brief document noting places where either:

- Requirements were not formalized (the 2% above),
- Additional detail was needed to formalize the requirements (due to some degree of ambiguity in the English documents), or
- Some choice was made between alternate interpretations of the written material (since the three English documents were written at different times, there were minor inconsistencies).

Once the requirements were formalized, we made and then encoded our design decisions. Again, there were decisions to be made about decomposition and abstraction. For each design decision, we needed to choose data structures for sorts and algorithms for operators. Specware™ comes with many built-in specifications that can be used for this (and other purposes). For example, there are specifications for sets, lists, and sets interpreted as lists. These design decisions then had to be verified to ensure that requirement properties

were upheld. Specware™ has a built-in resolution based theorem prover. This was used to prove roughly 25% of the proof obligations. Had we introduced intermediate lemmas (to essentially guide the automated theorem prover), this number would have been much higher. The other 75% of the proof obligations were proven by hand. Most of these proofs were relatively straightforward. Only the proofs required to establish the preconditions of the optimization algorithms were complicated enough to require actual use of pencil and paper. Eventually, every sort and operation had to be refined down into some data structure and operation provided by the Lisp programming language.

Once the software was initially generated, we maintained the software with Specware™. As we learned more about the problem domain, several changes were made to the requirement specifications, and the software was easily regenerated. None of these changes required significant redesign efforts, fortunately. However, one other change did. The initial optimization algorithm used an exhaustive search. For purposes of rapid prototyping, we had used the optimization problem theories (see figure 6) and search theories from the stiffened panel layout problem. Since these were inefficient, we encoded an additional theory of branch and bound optimization problems (see figure 7), and applied a corresponding search theory (see figure 8) to the domain of the equipment locator problem. The general branch and bound theories are completely reusable (i.e., independent of the domain in which they are instantiated).

A final comment is in order about the replay of software derivations in the presence of modified requirements. In attempting to replay a derivation, each proof obligation that was originally proven automatically by the theorem prover will be attempted again. Those that fail in the presence of modified requirements will be pointed out. For those proofs that were done by hand, a manual process of rechecking these proofs is required. At a minimum, each of these proof obligations can be pointed out to a designer. Nevertheless, a greater amount of automated proofs will give a higher assurance of the quality of the rederived software.

5.3. Level of effort and perceived benefit

We estimate that it took about:

- Four person-months to produce the formalized requirement specifications,
- Five person-months to produce the formalized design specifications (down to software), and
- One person-month to perform verification and validation (e.g., compare formalized requirements to informal requirements, run the theorem prover on refinement steps, and informally prove the remaining proof obligations).

In total, about ten months were spent on the entire process. We estimate that it would have taken about five months to produce the software using traditional approaches to software development. An additional month would have been spent in software testing.

Thus, our formal approach took almost twice as long. So what is the perceived benefit of this technology in this particular case? Presumably, the quality of the synthesized software is higher than it would have been had we used traditional approaches. Maintenance of

```
spec problem is
 sort D, R
 op I : D -> Boolean
 op O : D, R -> Boolean
 end-spec

%%% optimization problems, where one has:
%%% 1. inputs, outputs, and sets of outputs,
%%% 2. a predicate that tells when an input is valid,
%%% 3. a feasibility condition on (input,output) pairs, and
%%% 4. a cost function on outputs whose codomain is reals.

spec optimization-stuff is
 import translate colimit of diagram
  nodes triv, problem, set, real
  arcs triv -> problem : { E -> R },
      triv -> set : { E -> E }
  end-diagram
 by { D -> Input,
      E -> Output,
      Set -> Set-of-Output,
      I -> Valid-Input,
     O -> Feasible-Output }
 op cost : Output -> Real
 end-spec

spec optimization-problem is
 import optimization-stuff
 op optimal-output : Input, Output -> Boolean
 op optimal-solutions : Input -> Set-of-Output
 axiom (implies (valid-input input)
                (iff (optimal-output input output)
                   (and (feasible-output input output)
                        (implies (feasible-output input x)
                                 (leq (cost output) (cost x))))))
 axiom (implies (not (valid-input input))
                (iff (optimal-output input output) false))
 axiom (implies (valid-input input)
                (iff (in x (optimal-solutions input))
                   (optimal-output input x)))
 axiom (implies (not (valid-input input))
                 (equal (optimal-solutions input) empty-set))
 end-spec
```

Figure 6. Specifications for optimization problems.

the synthesized software was fairly easy, and had the specifications for branch and bound problems already existed, maintenance would have been nearly trivial. This latter comment gets at the heart of potential benefits of this technology.

Some benefits of this technology (i.e., faster and cheaper software development and maintenance) derive from the reuse of knowledge across time and applications. These

```
%%% Optimization Problems that are suitable for branch and
%%%  bound search strategies that find all optimal solutions

spec branch-and-bound-optimization-problem is

 import translate colimit of diagram
   nodes optimization-problem, ulist
   end-diagram  by { e -> description,
                     ulist -> ulist-of-description,
                     undefined-element -> undefined-description }

 const max-cost : real
 axiom (gt max-cost (cost output))
 op describes? : description, output -> boolean
 op initial-description : input -> description
 op branch : input, description -> ulist-of-description
 op bound : description -> real
 op leaf? : input, description -> boolean
 op extract : description -> output

 %%% the initial description describes all optimal outputs
 axiom (implies (valid-input input)
                (implies (optimal-output input output)
                         (describes? (initial-description input) output)))

 %%% branching on a description d results in mutually exclusive descriptions
 axiom (implies (valid-input input)
                (not (ex (output d1 d2)  (and (and (and (and
                         (member? d1 (branch input d))
                         (describes? d1 output))
                         (member? d2 (branch input d)))
                         (describes? d2 output))
                         (not (equal d1 d2))))))
 %%% that collectively describe all the feasible outputs described by d
 axiom (implies (valid-input input)
                (implies (and (describes? d output) (feasible-output input output))
                         (ex (description) (and (member? description (branch input d))
                                                (describes? description output))))))

 %%% bound is a lower bound on the costs of  all of the outputs described by description
 axiom (implies (describes? description output)  (geq (cost output) (bound description)))

 %%% a leaf description describes exactly one feasible output
 axiom (implies (valid-input input)
          (iff (leaf? input description)
             (and (ex (output) (and (describes? description output) (feasible-output input output)))
                  (implies (and (describes? description output1) (describes? description output2))
                           (equal output1 output2))))))
```

Figure 7. Specifications for branch and bound optimization problems.

(*Continued on next page.*)

```
%%% extract the output described by a leaf description
axiom (implies (valid-input input)
               (implies (leaf? input description)
                        (describes? description (extract description))))

%%% the remainder of this specification ensures algorithm termination

op valid-path-from-root? : input, ulist-of-description -> boolean
axiom (implies (valid-input input)
               (implies (null? uld)  (iff (valid-path-from-root? input uld) false)))
axiom (implies (valid-input input)
               (implies (nonnull? uld)
                        (iff (valid-path-from-root? input uld)
                             (and (equal (head uld) (initial-description input))
                                  (valid-path? input (initial-description input) (tail uld))))))
op valid-path? : input, description, ulist-of-description -> boolean
axiom (implies (valid-input input)
               (implies (null? uld) (iff (valid-path? input d uld) true)))
axiom (implies (valid-input input)
               (implies (nonnull? uld)
                        (implies (member? (head uld) (branch input d))
                                 (iff (valid-path? input d uld)
                                      (valid-path? input (head uld) (tail uld))))))
axiom (implies (valid-input input)
               (implies (nonnull? uld)
                        (implies (not (member? (head uld) (branch input d)))
                                 (iff (valid-path? input d uld) false))))
%%% Termination conditions:
%%% 1. all valid-paths-from-root must be finite
axiom (implies (valid-input input)
               (ex (n:nat)  %%% bound for lengths of all paths
                   (fa (uld:ulist-of-description)
                       (implies (valid-path-from-root? input uld) (lt (length uld) n)))))
%%% 2. branching produces finite number of children
%%% ... thus, there are a finite number of paths ...
axiom (implies (valid-input input)
               (ex (n:nat)  %%% bound for all branches
                   (fa (d:description)
                       (lt (length (branch input d)) n))))
%%% Thus, there are a finite number of search tree nodes

end-spec
```

Figure 7. (*Continued*).

```
spec branch-and-bound-algorithm is

 import branch-and-bound-optimization-problem

 definition of optimal-output is
 axiom (equal (optimal-output i o) (in o (cheapest-outputs i)))
 end-definition

 definition of optimal-solutions is
 axiom (equal (optimal-solutions i) (cheapest-outputs i))
 end-definition

sort solutions
sort-axiom solutions = set-of-output, real

op make-solutions : set-of-output, real -> solutions
definition of make-solutions is
axiom (equal (make-solutions s r) < s r >)
end-definition

op outputs : solutions -> set-of-output
definition of outputs is
axiom (equal (outputs s) ((project 1) s))
end-definition

op cost : solutions -> real
definition of cost : solutions -> real is
axiom (equal (cost s) ((project 2) s))
end-definition

%%% The algorithm follows...

op cheapest-outputs : Input -> Set-of-Output
definition of cheapest-outputs is
axiom (implies (valid-input Input)
               (equal (cheapest-outputs Input)
                      (outputs (bb-dfs input (initial-description input) (make-solutions empty-set max-cost)))))
axiom (implies (not (valid-input Input))
               (equal (cheapest-outputs Input) empty-set))
end-definition

op bb-dfs : input, description, solutions -> solutions
definition of bb-dfs is
axiom (implies (gt (bound d) (cost s))
               (equal (bb-dfs i d s) s))
```

Figure 8. Specifications for branch and bound search algorithm.

(*Continued on next page.*)

```
axiom (implies (leaf? i d)
               (equal (bb-dfs i d s)
                      (bb-dfs-leaf (extract d) (cost (extract d)) s)))
axiom (implies (and (not (leaf? i d)) (leq (bound d) (cost s)))
               (equal (bb-dfs i d s)
                      (bb-dfs-nonleaf i (branch i d) s)))
end-definition

op bb-dfs-leaf : output, real, solutions -> solutions
definition of bb-dfs-leaf is
axiom (implies (gt r (cost s))
               (equal (bb-dfs-leaf o r s) s))
axiom (implies (equal r (cost s))
               (equal (bb-dfs-leaf o r s)
                      (make-solutions (insert o (outputs s)) r)))
axiom (implies (lt r (cost s))
               (equal (bb-dfs-leaf o r s)
                      (make-solutions (singleton o) r)))
end-definition

op bb-dfs-nonleaf : input, ulist-of-description, solutions -> solutions
definition of bb-dfs-nonleaf is
axiom (implies (null? uld)
               (equal (bb-dfs-nonleaf i uld s) s))
axiom (implies (nonnull? uld)
               (equal (bb-dfs-nonleaf i uld s)
                      (bb-dfs-nonleaf2 (bb-dfs i (head uld) s) i (tail uld) s)))
end-definition

op bb-dfs-nonleaf2 : solutions, input, ulist-of-description, solutions -> solutions
definition of bb-dfs-nonleaf2 is
axiom (implies (gt (cost recur) (cost s))
               (equal (bb-dfs-nonleaf2 recur i uld s)
                      (bb-dfs-nonleaf i uld s)))
axiom (implies (equal (cost recur) (cost s))
               (equal (bb-dfs-nonleaf2 recur i uld s)
                      (bb-dfs-nonleaf i uld (make-solutions (union (outputs recur) (outputs s)) (cost s)))))
axiom (implies (lt (cost recur) (cost s))
               (equal (bb-dfs-nonleaf2 recur i uld s)
                      (bb-dfs-nonleaf i uld (make-solutions (outputs recur) (cost recur)))))
end-definition

end-spec
```

Figure 8. (*Continued*).

benefits can only be accrued over time. Once a large knowledge base of specifications has been built up (perhaps over several years), it is expected that development and maintenance times and costs will be greatly reduced. This remains to be demonstrated.

This poses a challenge for technology transfer within a corporation. Our current strategy is to focus on a "vertical market segment" of applications—in this case, the class of electrical engineering layout problems—and develop two or three applications, and their associated specifications (for both requirements and designs). Once this critical mass of knowledge has been developed, transferring this application development suite to a receiving organization should be more palatable to an organization with near term time horizons. In this way, the technology can be developed and transferred.

6. Getting the technology into broader use

Our overall impression is that this technology does work for industrial strength applications, but that it needs additional work to make it more usable. Various suggestions for technology and tool improvement are given in this section. While some of these suggestions apply to other formal software synthesis tools (e.g., the first three suggestions), some apply solely to Specware™.

6.1. Methodology for understanding refinement system interfaces

For maximum ease of use, the user interface of any system should reflect and reinforce the user's mental model of the artifacts and processes involved. One must keep in mind the education, experience, and general characteristics of the user community. It is one thing to have the underlying theory of a tool like Specware™ be based in category theory. It is quite another to have categorical terminology and concepts explicitly in the user interface of the tool. In an effort to have the tool be usable by a broader range of engineers, an attempt should be made to present things in the user interface that do not require explicit knowledge of category theory.

To improve the usability of this technology, human factors and usability engineers should perform task analysis, domain modeling, and requirement elicitation/engineering for the use of this technology by its intended audience. The emerging understanding of how this technology could be used should drive the requirements of the user interface.

6.2. Linking nonformal requirements to formal requirements

The people most familiar with application domains may not be interested or willing to author requirements in a language based on higher-order sorted logic and category theory. Natural languages are more widely known and accepted, and provide needed ambiguity. In addition, individual communities often have their own notations and/or visual symbology (this is quite typical in many traditional engineering disciplines). There needs to be some type of linkage between less structured representations of requirements and formal requirement specifications.

We have in mind a Web-based interface scenario in which engineers can click on a portion of a natural language document and get access to portions of the formal requirement

specifications. Perhaps restricted natural language grammars can be used to present the formal specifications in a more readable fashion.

6.3. Viewing linkage between requirements, design, and software

In the same vein as the previous suggestion, it would be nice to be able to click on portions of formal requirement specifications and get access to those portions of design specifications that reflect those requirements. These portions could be sorts, operations, axioms, or theorems. Similarly, clicking on portions of design specifications could lead to portions of other design specifications and/or software.

The theme in these two subsections is visibility and traceability of requirements and designs through the software derivation history. If someone else were to pick up our work on the equipment locator problem, what could be provided to them that would make it easier for them to understand how requirements and designs are achieved throughout the software derivation?

6.4. Better derivation replay and visibility

To replay the entire software derivation for the equipment locator problem takes about 30 minutes and 150 operations via the user interface. While it is probably not hard to do, the tool needs to better automate this task by recording the sequence of steps and then replaying them automatically.

There is structure to these sequences of steps, and how these sequences get put together (Jullig and Srinivas, 1993). This structure should be easily captured, displayed, and manipulated in the user interface of the tool. Note that this suggestion differs from the previous subsection in the granularity of what is being tracked. Here, we are tracking things at the level of specifications, diagrams, interpretations, etc. In the previous subsection, we are interested in portions of specifications (specific sets of statements/axioms, objects/sorts, operators, etc.).

6.5. Improved specification libraries

If larger specification libraries were available for a tool like Specware™, then people would be more productive with this technology. Some obvious examples would be material from data structure and algorithm courses; e.g., trees, graphs, tries, search algorithms, disjoint set-union-find algorithms, etc. Other suggestions include theories for dimensional analysis and units conversion of physical quantities (e.g., the engineering math ontology of (Gruber et al., 1994)), theories of geometry, and theories of basic material properties. We have done work in the first two areas.

6.6. Better proof support

The effective use of the automated theorem prover in Specware™ is limited. One way to improve this would be to use different automated theorem provers, equation solvers, or

model checkers, individually or perhaps in some combination. Another, more pragmatic approach might be to use a combination of proof checking and automated theorem proving. As a software designer, as I do my proofs "by hand", why not allow me to record my own proof steps? Perhaps a proof checker can validate some proof steps fairly easily. Perhaps some steps require a fair amount of deduction, but might be possible to discharge in an interactive manner. In the worse case, simply allow me to record my own "proof," which can be manually verified during maintenance operations.

6.7. Generating state based programs

The current version of Specware™ only generates software that is purely functional. No use of state based variables is possible. This limits the efficiency of generated software and precludes Specware's use in embedded systems, where the improved quality of generated software is highly desirable.

6.8. Software optimization transformations

The KIDS system (Smith, 1991) has some extremely useful software level optimization transformations. These range from low-level transformations that are similar to compiler optimizations to high-level transformations that perform finite differencing. Some of these capabilities exist in Specware™, but many are not yet generally available. To get more efficient software, these capabilities are needed.

6.9. Expressing rationale behind design choices

During design there are often multiple design alternatives. It would be nice if we could clearly record these alternatives and the non-functional rationale behind the current selection of a specific design. This type of rationale could even be stated formally; e.g., in terms of trade-offs between space and time complexities. When changes in requirements happen over time, this information would be very valuable. All that is currently stored in a derivation history is one way to achieve a software solution to a problem. As one initially explores the design space/alternatives, one learns a lot of information about various trade-offs. Let's capture that information, and leverage it in the future.

6.10. Refinement system interface tailorability

As mentioned earlier, it is desirable to create a vertical market segment capability oriented toward synthesizing and maintaining software in a family of related applications. For maximal ease of use, it is possible and desirable to create a front-end user interface to Specware™ that is tailored to this specific domain (Smith, 1999). In order to do this, an application program interface (API) to Specware™ is needed. This API needs to be supported and clearly documented.

7. Requirements elicitation

Systems development and engineering does not often begin with a clear and formal statement of requirements. A system such as Specware™ allows reuse of knowledge from that point forward, but preceding this, are there opportunities for knowledge reuse? If so, can one find a common underlying theory for knowledge structuring and reuse that allows an integration of requirements elicitation techniques with the system-development techniques describing previously in this paper? These are some of the questions underlying our current work.

7.1. Human factors engineering

The most common causes of failure of software development projects are a lack of good requirements and a lack of proper understanding of users' needs (www.standishgroup.com). Customers and users are domain experts, not requirements development experts. In many project meetings, software architects say, "The customers don't know what they want." They do know what they want. They just don't know how to tell you. In addition, users' statements of their needs are highly unstructured and difficult to formalize. To make matters even more challenging, some of the domain knowledge may exist only as collective knowledge of a group of people. This "tribal" knowledge is rarely written down, let alone formally encapsulated.

Our approach to requirements elicitation is based on human factors engineering techniques and processes for eliciting users' task knowledge while capturing the unstructured domain knowledge as formal models of the users' cognitive tasks. These techniques and processes are outlined in Mayhew (1999) and Rogers (1992). While the models of users' cognitive tasks are formal, the techniques and processes for abstracting system requirements from these models are not. An initial framework for structuring human factors design knowledge is depicted in figure 9.

The ability to represent domain knowledge as a user interface allows us to do two key things. First, using a user interface prototype as a communication tool can iteratively refine our knowledge of the domain. User and domain experts often have difficulty expressing highly technical domain knowledge in terms an expert in another domain, such as software engineering, can grasp. Second, the user interface can be used to allow the users and domain experts to interact with the to-be system (albeit in a limited way) to perform common tasks. Customers can then "buy-off' on the to-be system since, to the customers, domain experts, and users, the user interface **is** the system (Mayhew, 1999). Buying off on the system, before system construction, is very effective at stabilizing requirements. A major cause of high software maintenance costs is "changing customer requirements." Closer examination shows that the customer didn't change requirements—the requirements were poorly understood until system delivery when the user first could get hands-on experience with the system.

7.2. Algebraic semiotics

We propose that category theory provides a mathematical foundation for translating user task models and domain knowledge into formal system requirements. Possible techniques

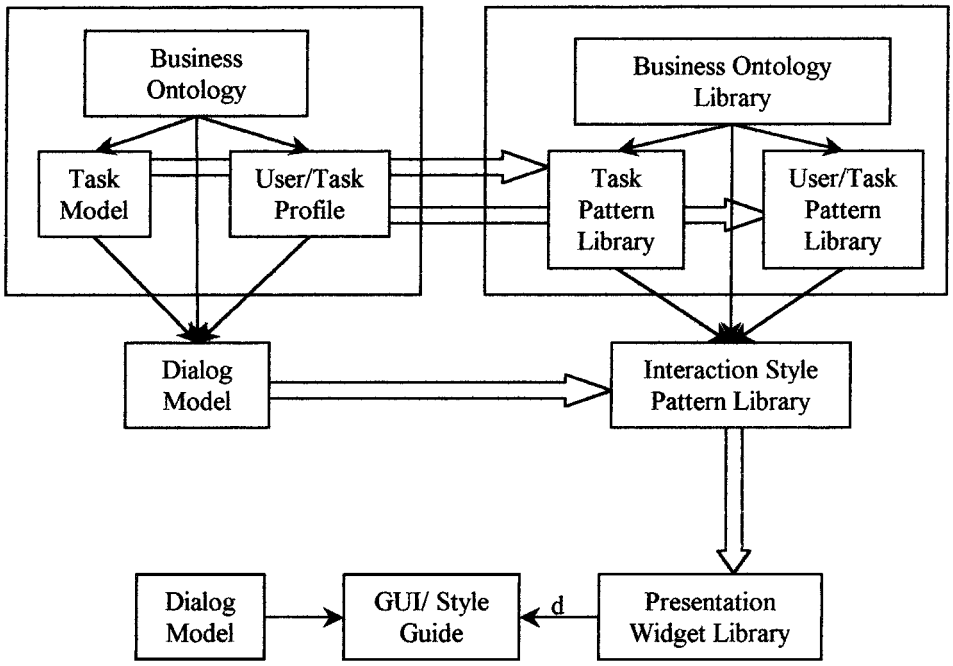

Figure 9. Framework for structuring human factors design knowledge.

for doing this have been proposed under the term algebraic semiotics (Goguen, 1999). Semiotics is the study of systems of symbols and their meaning. Symbol systems include, for example, Object Oriented class models, user interfaces, task models, and even the users' internal cognitive representation of domain knowledge (i.e., mental models (Rogers et al., 1992)). Formalizing these representations of knowledge in category theoretic systems allows a well-defined, mathematical approach to understanding reuse of engineering domain knowledge at higher levels of abstraction.

The process for moving from domain knowledge to software proceeds through several discrete stages. The highly informal and unstructured domain knowledge that exists in the minds of domain experts and users is represented as a mental model. The mental model can be represented as a symbol system whose structure is defined in category theory. The users' interface to the system is some translation of the users' mental model, and is itself a symbol system. The symbol systems as well as the translations are represented in category theory (Goguen, 1999). The groupings of controls and displays in the user interface, and the order in which users access them to perform tasks, determine the information processing demands imposed on the system by the user. This defines the system requirements for most business systems. This does not mean that all aspects of system functions are in the user interface, only that the need for the system functions is implied by the user interface. For example, in the equipment locator problem above, the formula for calculating RFI is not in the user interface. However, the user interface does show the need for making this calculation, and the requirement is defined.

8. Summary

We have described our experiences in applying a category theory based approach to industrial strength software specification, synthesis, and maintenance. This paradigm is one that allows the capture and structuring of formal requirement specifications, design specifications, implementation software, and the refinement processes that lead from requirements to software. In this approach, the refinement process can guarantee correctness of the generated software. By recording, modifying, and then replaying the refinement history, we are able to more easily maintain the software. By capturing, abstracting, and structuring knowledge in a modular fashion, we are able to more easily reuse this knowledge for other applications.

Our overall impression is that the technology does work for industrial strength applications, but that it needs additional work to make it more usable. It is interesting to note that when this technology is applied to software systems whose outputs are designs for airplane parts, the design rationale that is captured is not only software engineering design rationale, but also design rationale from other, more traditional, engineering disciplines (e.g., mechanical, material, manufacturing, electrical, human factors, etc.). This suggests the technology provides an approach to general systems engineering that enables one to structure and reuse engineering knowledge broadly.

References

Bjorner, D. and Jones, C. 1982. *Formal Specification & Software Development*. Prentice-Hall International, Englewood Cliffs, New Jersey.

Blaine, L. and Goldberg, A. 1991. DTRE—A semi-automatic transformation system. In B. Moller, editor, *Constructing Programs from Specifications*. North Holland, New York, New York.

Burstall, R.M. and Goguen, J.A. 1980. The semantics of clear, a specification language. In *Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification. Lecture Notes in Computer Science*, vol. 86, Springer-Verlag.

Crole, R. 1993. *Categories for Types*. Cambridge University Press, New York, New York.

Gannon, J. et al. 1994. *Software Specification—A Comparison of Formal Methods*. Ablex Publishing, Norwood, New Jersey.

Goguen, J.A. 1970. Mathematical representation of hierarchically organized systems. *Global Systems Dynamics.* In E. Attinger and S. Karger, editors, pp. 112–128.

Goguen, J.A. 1999. An introduction to algebraic semiotics, with applications to user interface design. In C. Nehaniv, editor, *Computation for Metaphor, Analogy and Agents. Springer Lecture Notes in Artificial Intelligence.*

Goguen, J.A. and Burstall, R.M. 1992. Institutions: Abstract model theory for specification and programming. *Journal of the Association of Computing Machinery*, 39(1):95–195.

Gruber, T. et al. 1994. An ontology for engineering mathematics. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*. Morgan Kauffman.

Human Engineering Program—Processes and Procedures. 1996. US Department of Defense, Handbook MIL-HDBK-46855A.

Jullig, R. and Srinivas, Y.V. 1993. Diagrams for software synthesis. In *Proceedings of the 8th Knowledge-Based Software Engineering Conference*. Chicago, IL.

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

Mayhew, D.J. 1999. *The Usability Engineering Lifecycle*. Academic Press/Morgan Kauffman, New York, New York.

Meseguer, J. 1989. General logics. In Ebbinghaus et al., editors, *Logic Colloquium '87*. Elsevier Science Publishers, New York, New York.

Pierce, B.C. 1994. *Basic Category Theory for Computer Scientists*. MIT Press, Cambridge, Mass.

Rogers, Y. et al. 1992. *Models in the Mind—Theory, Perspective, and Application.* Academic Press, New York, New York.

Smith, D. 1991. KIDS: A knowledge based software development system. In M. Lowry and R. McCartney, editors *Automating Software Design.* MIT Press, Cambridge, Mass.

Smith, D. 1991. Mechanizing the development of software. In M. Broy, editor, *Calculational System Design.* NATO ASI series, IOS Press.

Spivey, J.M. 1992. *The Z Notation: A Reference Manual.* Prentice-Hall, New York.

Srinivas, Y.V. and Jullig, R. 1995. Specware^{TM}: Formal support for composing software. In *Proceedings of the Conference of Mathematics of Program Construction*. Kloster Irsee, Germany.

Waldinger, R. et al. 1996. *Specware^{TM} Language Manual 2.0.1*, Suresoft, Inc.

Wang, T.C. and Goldberg, A. 1991. A mechanical verifier for supporting the design of reliable reactive systems. In *International Symposium on Software Reliability Engineering*, Austin, Texas.

Williamson, K. and Healy, M. 1997. Formally specifying engineering design rationale. In *Proceedings of the Automated Software Engineering Conference.*

Williamson, K. and Healy, M. 2000. Deriving engineering software from requirements. *International Journal of Intelligent Manufacturing*, Kluwer Academic Publishers, Hingham, MA, 11(1):3–28.