Industrial Applications of Software Synthesis via Category Theory

Keith Williamson, Michael Healy
The Boeing Company, Seattle, Washington
Keith.Williamson@boeing.com

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, etc.). This suggests the technology provides an approach to general systems engineering that enables one to structure and reuse engineering knowledge broadly.

1. Introduction

Industry demands that software development and maintenance be made faster, cheaper, and better. Conspiring against this are high labor turnover rates in the software industry, and increasingly in American industry more broadly. Institutional memory is being lost in this turnover. Is there a way to capture, structure, use and then reuse this knowledge for the purposes of software development and maintenance?

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.

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 Specware™.

2. Formal Preliminaries

The field of category theory [4,10,12] provides a foundational theory. This theory was applied to systems theory and systems engineering [3,6]. This theory was embodied in the software development tool Specware™ [14,16,17].

2.1. Category of Signatures

A *signature* consists of the following:

1. A set S of sort symbols
2. A triple O = < C, F, P > 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 = < S, O >, and
2. A set Ax of axioms over Sig

Given two specifications <Sig1, Ax1> and <Sig2, Ax2>, a signature morphism M between Sig1 and Sig2 is a *specification morphism* between the specifications iff:

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

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 translation). Of course, Ax2 may be a stronger theory. The category **Spec** consists of objects that are specifications and morphisms that are specification morphisms.

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™ 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 1 and 2 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 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.

The colimit operation can be used to compose any specifications, which can represent problem statements, theories, designs, architectures, or programs. The fact that specifications can be composed via the colimit operation allows us to build specifications by combining simpler components modularly, just as 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 [13], REACTO [18], and DTRE [2]), 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 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 [15], VDM [1], and Larch among many others [5]. 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 [7,11] 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 [17], which helps with the scalability of the technology. Multiple categories underlie Specware™.

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 [19]. 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. 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) 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 design rationale and tie it directly to the software. For this purpose, we used the Specware™ system to first document the structural and manufacturing 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).

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 1). 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 2). This specification was then imported into another, which added manufacturing properties 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 [20].

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 [20]. 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.

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™ 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. 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 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 the formal specifications, here are three of the hard constraints (Specware™ does have a prefix notation, but we did not use it):

```
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))))))))
```

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 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. The other 75% were proven by hand. Eventually, every sort and operation had to be refined down into some data structure and operation provided by the Lisp programming language.

Finally, 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 3) and search theories from the stiffened panel layout problem. Since these were inefficient, we encoded an additional theory of branch and bound optimization problems, and applied a corresponding search theory 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).

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.

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, it 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 [9]. 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 unit conversion of physical quantities (e.g., the engineering math ontology of [8]), theories of geometry, and theories of basic material properties. We have done some 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. This limits Specware's use in embedded systems, where the improved quality of generated software is highly desirable.

6.8. Software Optimization Transformations

The KIDS system [13] has some very nice, and extremely useful, program level optimization transformations. These range from low-level transformations that are similar to compiler optimizations, to high-level transformations that perform finite differencing. 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 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.

7. 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, etc.). This suggests the technology provides an approach to general systems engineering that enables one to structure and reuse engineering knowledge broadly.

8. Bibliography

[1] Bjorner, Dines and Jones, Cliff, *Formal Specification & Software Development*, Prentice-Hall International, 1982.

[2] Blaine, Lee and Goldberg, Allen, DTRE – A Semi-Automatic Transformation System, in *Constructing Programs from Specifications*, ed. B. Moller, North Holland, 1991.

[3] Burstall, R. M. and Goguen, J. A., The Semantics of Clear, a Specification Language, in Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification, *Lecture Notes in Computer Science, 86*, Springer-Verlag, 1980.

[4] Crole, Roy, *Categories for Types*, Cambridge University Press, 1993.

[5] Gannon, John et al., *Software Specification - A Comparison of Formal Methods*, Ablex Publishing.

[6] Goguen, J. A., Mathematical Representation of Hierarchically Organized Systems, in *Global Systems Dynamics*, ed. E. Attinger and S. Karger, 1970, pp. 112-128.

[7] Goguen, J. A. and Burstall, R. M., Institutions: Abstract Model Theory for Specification and Programming, *Journal of the Association of Computing Machinery*, 1992.

[8] Gruber, Tom et al., An Ontology for Engineering Mathematics, in Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning, Morgan Kauffman, 1994.

[9] Jullig, R. and Y. V. Srinivas, Diagrams for Software Synthesis, Proceedings of the 8th Knowledge-Based Software Engineering Conference, Chicago, IL, 1993.

[10] MacLane, Saunders, *Categories for the Working Mathematician*, Springer-Verlag, 1971.

[11] Meseguer, Jose, General Logics, *Logic Colloquium '87*, Eds. Ebbinghaus et al., Elsevier Science Publishers, 1989.

[12] Pierce, Benjamin C., *Basic Category Theory for Computer Scientists*, MIT Press, 1994.

[13] Smith, Doug, KIDS: A Knowledge Based Software Development System, in *Automating Software Design*, Eds. M. Lowry and R. McCartney, MIT Press, 1991.

[14] Smith, Doug, Mechanizing the Development of Software, in *Calculational System Design*, Ed. M. Broy NATO ASI series, IOS Press, 1999.

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

[16] Srinivas, Y. V. and Jullig, Richard, Specware™: Formal Support for Composing Software, in Proceedings of the Conference of Mathematics of Program Construction, Kloster Irsee, Germany, 1995.

[17] Waldinger, Richard et al., *Specware™ Language Manual 2.0.1*, Suresoft, Inc, 1996.

[18] Wang, T. C. and Goldberg, Allen, A Mechanical Verifier for Supporting the Design of Reliable Reactive Systems, International Symposium on Software Reliability Engineering, Austin, Texas, 1991.

[19] Williamson, K. and Healy, M., Formally Specifying Engineering Design Rationale, in Proceedings of the Automated Software Engineering Conference, 1997.

[20] Williamson, K. and Healy, M., Deriving Engineering Software from Requirements, *Journal of Intelligent Manufacturing*, to appear, 1999.

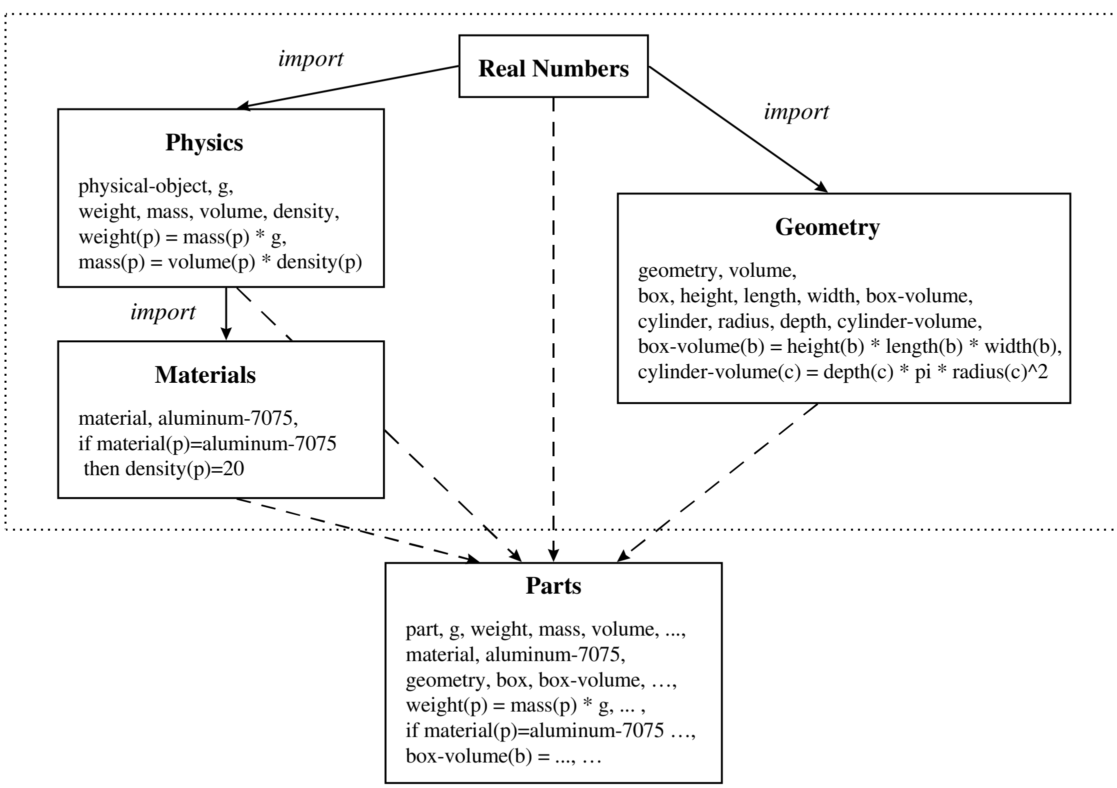

Figure 1. Colimit of a Specification Diagram

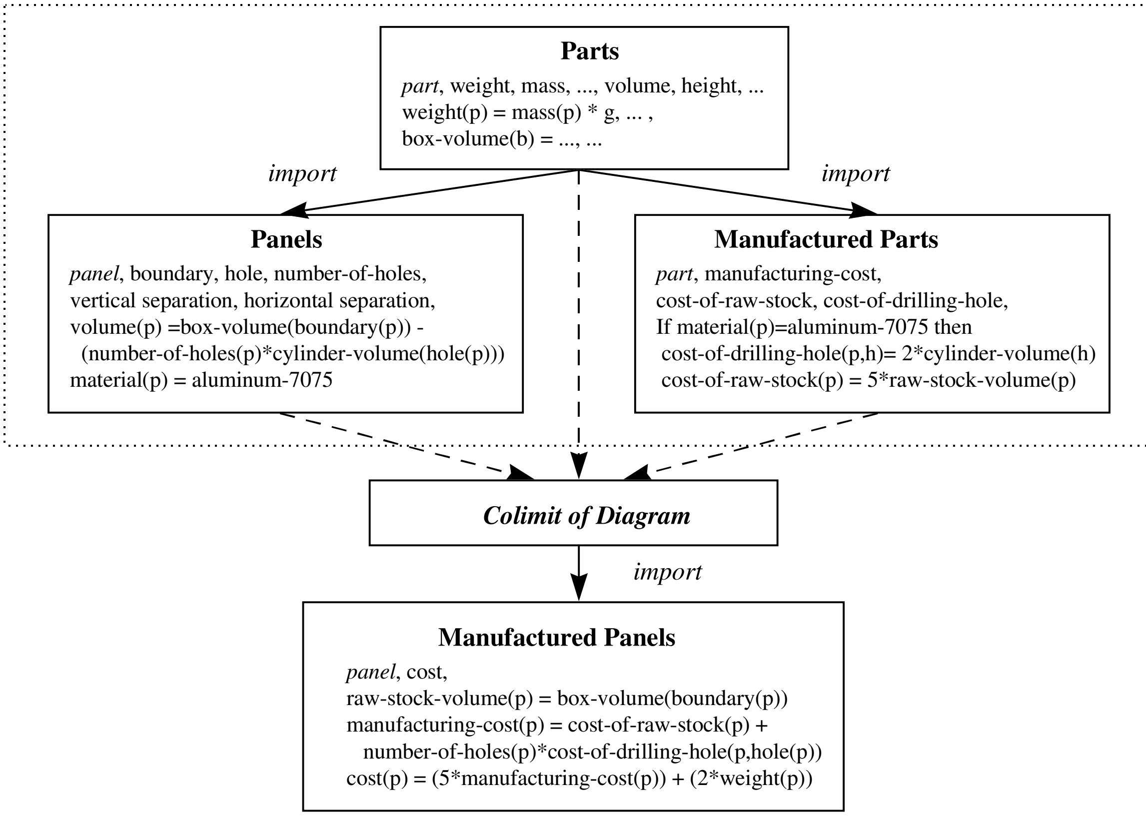

Figure 2. Another Colimit and Specification Morphism

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

diagram prob-set-real-diagram is
  nodes triv, problem, set, real
  arcs triv -> problem : { E -> R },
       triv -> set : { E -> E }
  end-diagram

spec optimization-stuff is
  import translate colimit of prob-set-real-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))
            (not (optimal-output input output)))
  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 3. Some Specifications for Optimization Problems