Applying Category Theory to Derive Engineering Software from Encoded Knowledge

Michael Healy[1] and Keith Williamson[2]

[1] The University of Washington,
co 13544 23rd Place NE, Seattle,Washington 98125, USA
mjhealy@u.washington.edu
[2] The Boeing Company, PO Box 3707 MS 7L-66, Seattle, Washington 98124, USA
keith.e.williamson@boeing.com

Abstract. In an industrial research project, we have demonstrated the feasibility of applying category-theoretic methods to the specification, synthesis, and maintenance of industrial strength software systems. The demonstration used a first-of-its-kind tool for this purpose, Kestrel's Specware™ software development system. We describe our experiences and discuss broadening the application of such category-theoretic methods in industry.
Although the technology is promising, it needs additional development to make it generally usable. This is not surprising given its mathematical foundation. On the other hand, we believe our demonstration is a turning point in the use of mathematically rigorous approaches in industrial software development and maintenance. We have demonstrated here the capture via mathematical methods not only of software engineering design rationale, but also of the product design and manufacturing process rationale used by different engineering disciplines, and the production of usable software directly from the captured rationale. We feel that that further evolution of the tools for this technology will make formal systems engineering a reality.

1 Introduction

Industrial software developers are operating under an increasingly challenging workload. Their clients want development and maintenance of large and business-critical software systems made faster, cheaper, and better. Conspiring against this are high turnover rates in software analysts and engineers, the creators and implementors not only of software systems but of the institutional knowledge they represent. Is there a way to capture insititutional knowledge efficiently and reliably in software development, and make it transparent and easily reusable so that system maintenance and use is less dependent upon the memories of individuals, system environment changes can be accomodated without the present major disruptions, and system knowledge can flow across organizational boundaries and through time free of the brittleness and inflexibility of legacy software?

T. Rus (Ed.): AMAST 2000, LNCS 1816, pp. 484–498, 2000.
© Springer-Verlag Berlin Heidelberg 2000

We have been experimenting with a new approach to addressing these questions. Transparency and flexibility of the new approach accrues from the mathematical rigor upon which it is based. This statement would seem paradoxical to the industrial software developers who are the desired users of our approach. Most are highly skeptical of formal methods because of their perceived opacity and INflexibility, together with the lack of success stories accompanying their use. Strong data typing and the requirement of deciphering program code to form provable assertions in a formal language are regarded with disdain: These activities are time-consuming and produce nothing. The developers, meanwhile, are under increasingly intense pressure to produce working software that helps build airplanes. The formal methods advocate must demonstrate an approach that helps accelerate the process by preventing errors upstream in the work flow, where most of the errors begin. The approach must do more, however. To avoid taxing the patience of developers under pressure to produce, it must compensate for the time spent using it by accelerating the later stages of software production. This must be demonstrated time and again, in projects of increasing complexity and magnitude, to convince the skeptics that the resulting methodology is indeed "industrial strength".

What we hope to establish is that constructive methods are available, developed on the mathematical foundation of category theory. As a result of the mathematical foundation, software development and maintenance can become a true engineering occupation, the engineering of industrial knowledge. On the other hand, the tools involved are at an early stage of development and the mathematics behind them must undergo further translation. Category theory, even more than most formal methods, can occupy a dedicated student for years. There is a great deal of research ahead in making our approach fully practicable across industry.

In this paper, we describe our experiences in applying a category-theoretic approach to software specification, synthesis, and maintenance. In addition, we share an industrial perspective on what is needed to broaden the appeal of this technology to industry. We begin with some formal preliminaries and a brief discussion of `Specware`™ [11], the category-theory-based tool we have been using.

2 Formal Preliminaries

Recently, category-theoretic methods have been applied directly in software development. The methodology described here is based upon a categorical approach introduced in [6]. The basic idea is to refine formal specifications into software by operations performed upon diagrams in certain categories, chiefly the automatic derivation of colimits. Earlier work established the theoretical background for this approach[4,5,7]. Several categories are involved here, beginning with the category of formal specifications and their morphisms. We express the many different aspects of programming in specifications of theories written in formal logic. The morphisms express the part-to-whole relationships of simple theories

to more complex ones by mapping the terms, sorts, and axioms of the simpler to the more complex theory. A basic property of a morphism is that the axioms of the domain of a morphism are theorems of the codomain (here, we are speaking of theories and their specifications interchangeably, taking some expository license with the formalisms used). The composition of morphisms then allows us to transition from abstract theories to constructive, complex theories in a succession of diagram refinement steps employing colimits. The colimits "put everything together", constructing a single theory from its component theories and their relationships.

A functorial, colimit-preserving mapping of diagrams into a category of programming language constructs allows the automatic construction of software as one end-item of this process, but there is more. The theories, morphisms and diagrams provide a valuable infrastructure for software development, maintenance, re-use and re-engineering. Should users' requirements change or the operating environment of an application vary across platforms and engineering and manufacturing organizations, we can make the required changes to the software by replaying the diagram refinement with some theories and morphisms added or replaced.

2.1 Commutative Diagrams and Colimits

A diagram in a category is a means of highlighting some objects and morphisms between them for special consideration. For example, we can obtain a theory of points as pairs of integers by combining a theory of pairs and a theory of integers along the common theory `Triv`. The theory `Triv` contains nothing but a sort symbol, and the morphisms from it to the other two theories of the base diagram for integer-points represents the fact that the sort of integers is the same as the sort from which pairs are formed in the put-together theory (Figure 2.1).

The colimit of a diagram is a cone, with morphisms from each diagram object to an apical object. The diagram so formed (the defining diagram, where the original diagram is called the base diagram), with the cone added to the original diagram, is commutative. Because of this, any specification T_0 of the base diagram which is "shared" by two or more other specifications T_i of the base diagram, with morphisms $m_i: T_0 \longrightarrow T_i \quad (i \neq 0)$, becomes mapped into the apical object in only one way, since any two compositions along paths from it to the apical object are equal. Further, a colimit is an initial cone for its base diagram, so that there is a unique morphism from its apical object to the apical object of any other cone for the same base diagram with the two cones together forming a commutative diagram. The significance of this is that the theory represented by the colimit apex is minimal with respect to the objects and morphisms in the base diagram: No extraneous information is included. Thus, we can build theories with a clear semantics in terms of their component theories and the relationships among the components. We can re-use such theories by including them in different ways in diagrams and taking the colimits, as indicated in Figure 2.1 where the theory of points is used to make theories of lists of points and sets of points.

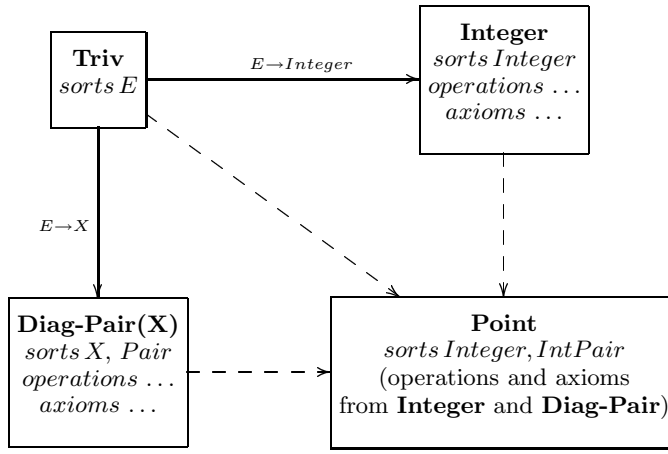


Fig. 1. Deriving a specification for points as integer-pairs. The colimit object contains the specifications for integers and pairs "welded together" so that the paired sort is the sort of integers.

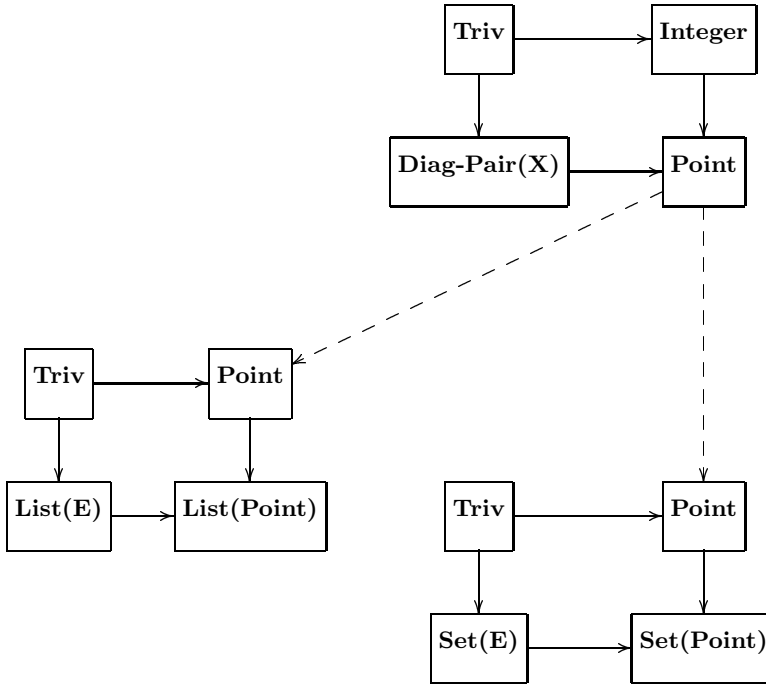


Fig. 2. Example of specification reuse: Colimit specification for points derived as pairs of integers, then re-used in further colimit derivations of lists of points and sets of points.

In our applications, we have colimit diagram derivations such as that in Figure 3. With such derivations, we combine domain theories, algorithm theories, and others to express the semantics of a software program as a construction from elementary "semantic modules" which are relatively easy to verify for correctness. The mathematical rigor of the colimit construction guarantees (in principle at least) that the combination is as correct as its components and their morphisms. As an example, we combine a theory of material properties, formed by adding definitions and axioms to a simple theory of physical properties (volume, mass, etc.) with a simple theory of geometry along a common theory of real numbers[1]. In the colimit apex, a theory of airplane parts, there is only one copy of the real numbers, since they can be embedded in the `Parts` theory in only one way despite their having been embedded separately in the theories of physical quantities and geometry in the original diagram. The colimit is calculated by an algorithm that implements coproduct-coequalizer derivations (based upon the dual to Theorem 1.9.7 in Pierce[8]).

To date, we have produced two industrial applications. The first [13,14] was a computer-aided design (CAD) software code for a certain kind of airplane part, a panel optimized for weight subject to loads and other design constraints—and manufacturing constraints as well (see Figure 4). The software is meant to operate within an intelligent CAD system, accepting panel sizing information as input and generating a panel design with holes drilled to save weight subject to the constraints. Aside from automated code generation, this first application allowed us to demonstrate an immediate advantage of the use of formal methods in software design: The engineering and manufacturing rationale that determined the software design was clearly and unambiguously expressed in the theories and morphisms that formed the refinement constructs. We were able to further demonstrate another major advantage: The refinement process which produced the software could easily be "re-played" to re-generate the software after changing the assumptions used to characterize the application. This was done with a single change to a manufacturing process theory in a diagram that appears later than that of Figure 3 in the refinement process. The resulting software code optimized the airplane part using a different objective, based upon changed process assumptions. This was our proof-of-concept demonstration.

The follow-on challenge to our proof-of-concept demonstration was a scale-up demonstration. Many industrial applications are a great deal more involved than the simple CAD application, resulting in an order of magnitude more code. We chose to produce a program for solving a discrete optimization problem—placing electrical components within a commercial airliner. Problem constraints include electrical component bay locations, and shelf and supporting equipment locations within the bays. There are many others—for example, many components are actively air-cooled, so must be placed near air-cooling equipment. Redundancy is an important factor in airplane design; redundant components must not be located in proximity to one another. The problem objective is to minimize

[1] The "geometry" and "real numbers" theories were produced purely for expediency in this application and lack mathematical elegance.

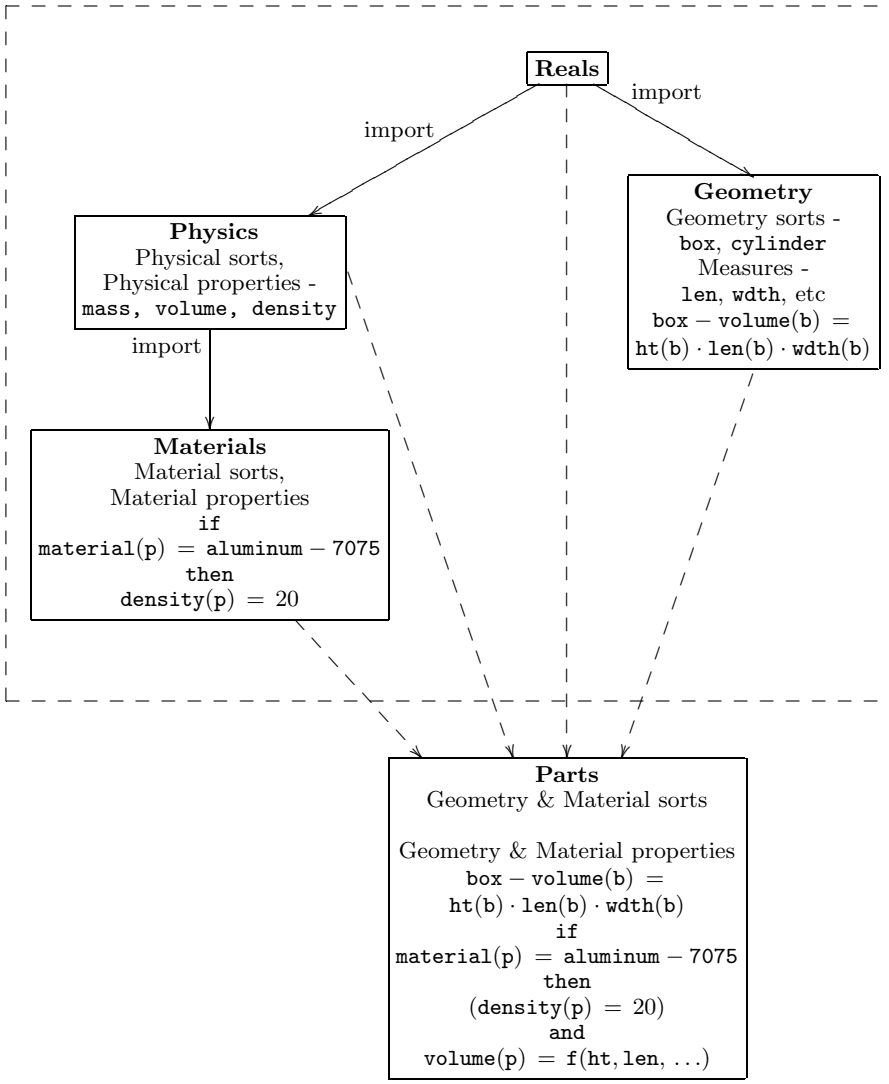


Fig. 3. A colimit derivation for a theory of airplane parts, combining theories of materials and geometry along the common theory of reals. Here, the theory of reals is a convenient abstraction of the real datatype often appearing in programming languages. The theories of materials and geometry used here are similar. Mathematically correct theories (or "first principles") would require more time to derive but would be better for re-useability.

the total length of wire needed to connect the components. This optimization problem is quite complex and good solutions are critical, making this a truly "industrial strength" application. Airplane layout and equipment placement design know-how were encoded as formalized knowledge in domain theories. Optimization problem-solving knowledge and the problem to be solved (i.e., the requirements) made up other theories. The following theories, with appropriate morphisms, made up the refinement process:

* Geometry[2],
* 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,
* A theory of optimization based upon branch and bound.

Roughly 7,000 lines of requirement and design specifications are needed to specify the statement and solution of this problem. The generated (Lisp) software exceeds 7,000 lines.

Again, we demonstrated not only refinement to a finished code based upon clearly-expressed design rationale, but also the ability to re-generate the code to meet changed requirements after making spot changes to component theories. This time, we were in a position to lay claim to a unique in-house capability to produce "industrial-strength" software with a methodology having the stated advantages.

This has turned out to be insufficient to guarantee implementation of this methodology. We shall discuss the reasons for this following a brief description of the tool we used.

3 The Software Development Tool Used

The tool we currently use for the categorical software development process is `Specware`™, a software development and maintenance environment supporting the specification, design, and semi-automated synthesis of software. `Specware`™ represents a confluence of capabilities and lessons learned from Kestrel's earlier prototype systems for software development, KIDS[9], REACTO[12], and DTRE[2]. It differs from the earlier systems in having a fully formal basis in category theory. It supports the automation of:

[2] Again, expedient.

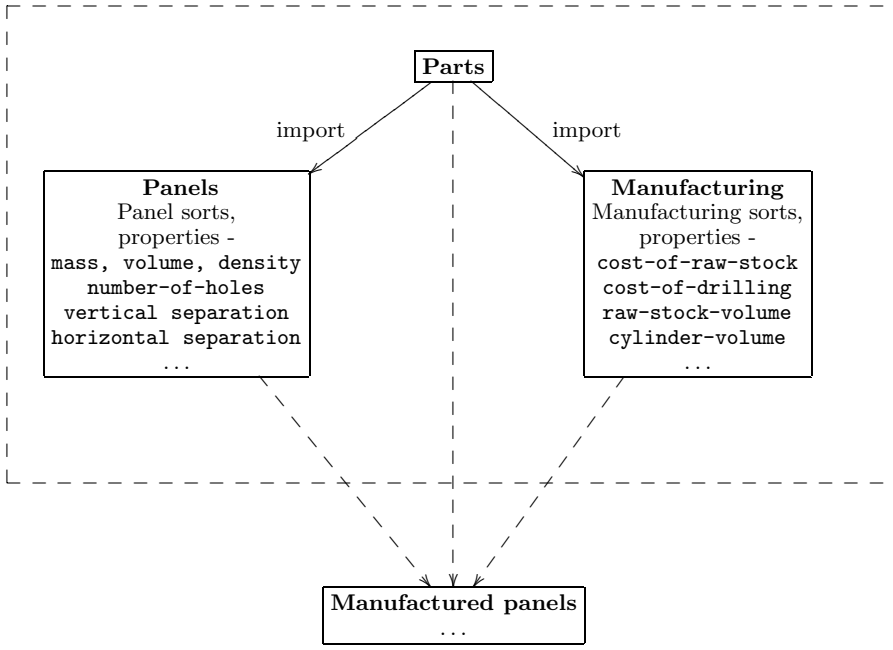


Fig. 4. A colimit derivation for a theory of manufactured panels, combining theories of panels and manufacturing along the common theory of parts.

* Component-based specification of programs using a graphical interface
* Incremental refinement of specifications into correct code in various target programming languages (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 community. There are numerous formal languages that have been used for specifying software systems, among them Z[10] and the Vienna Development Method (VDM) [1]. Of the many formal specification languages, 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 discharge (sometimes manually)

of "proof obligations" that arise in the refinement of a specification from one level to another.

4 Getting the New Technology into Routine Use in Industry

We can state an important principle of technology transfer in industry based upon upon many person-years of experience (ours and other researchers' as well): All the effort in obtaining funding for, doing the research and development on, and demonstrating a new technology seems small in comparison to the daunting task of getting it into everyday use.

Resistance comes from many quarters. Implementing a new technology generates a cost spike that attracts unfavorable attention. We need mention only the cost of training engineers and technicians to use a new technology based upon "esoteric mathematics" (in our case, algebraic methods in the form of category theory). The mathematical background of most industrial engineers encompasses, at most, second-year calculus, a semester of ordinary differential equations and a class in complex variables—no discrete mathematics, no abstract algebra and certainly no category theory. Resistance can come directly from technical people who have difficulty with the notion that they must learn a "new mathematics". In fact, we find that most computer science graduates hired by industry (including PhDs) have less in their mathematical backgrounds than most engineers—and do not readily agree to improve! In fact, one might ask: How many computer science graduates actually spend much time with discrete mathematics in today's curriculum, and how well does the average discrete mathematics course address the kinds of mathematics involved in the various algebraic methodologies? We think the answers are very discouraging for the adoption of mathematically-based software development methods. And, finally, training alone will not solve the difficulties in using a software tool that does not, in fact, have a relatively clear and straightforward usage. The ultimate resistance to introduction of a new technology occurs "in the trenches"—with the people who must use it in their everyday work but find it too difficult and time-consuming to be of practical use.

In the following, we shall cover areas in which our technology can be improved and made more amenable to technology transfer. We shall express much of this in terms of the particular tool we used in our effort. First, we present a brief discussion of our work flow.

5 Work Flow in Our Trial Applications

After having spent some time learning some of the basics of category theory as used in this methodology, we went on to learn how to use `Specware`™ while working on the panel layout problem. We also experimented with development of theories closer to pure mathematics, with a view to eventually developing

theories based upon "first principles" for physics, geometry and other areas that might lead into applications. We feel that such theories are essential for true re-use of the theory base at the earlier stages of refinement, where re-use can occur most effectively. Following our initial work, we went on to our second application, the "industrial strength" equipment locator problem.

We first proceeded to learn about the domain. We had three informal 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 we understood the requirements, we formalized them. We captured roughly 98% of the requirements found in the informal material. The remaining 2% dealt with the interfacing of the software system we were to generate with other software systems. At that time, we had insufficient information for dealing with such issues.

Following our initial formalization we performed a manual validation in which we compared the formal requirements with the informal ones. We wrote a brief document noting places where any of the following held:

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

Having documented these issues, we made and then encoded our design decisions. This involved further decomposition and abstraction. For each design decision, we needed to choose data structures for sorts and algorithms for operators. `Specware`™ has many built-in specifications that can be used for this and other purposes. This was of enormous help in beginning the creation of an infrastructure of theories and morphisms for re-use in many applications. Included are specifications for sets, lists, and sets interpreted as lists, all helpful in developing abstract data structures for software design. The design decisions needed verification to ensure that requirement properties were upheld. Fortunately (again), `Specware`™ has a built-in theorem prover. This was used to prove roughly 25% of the proof obligations. The other 75% were proven by hand. Eventually, we refined every sort and operation from abstract theoretical construct down to some implementable data structure or operation provided by the the target programming language (Lisp was the one we used—the `Specware`™ version we had acces to also had C++ implemented, but this did not always work with our compiler and was less straightforward to the one of us (Williamson) doing the bulk of the work).

Once an initial pass at generating the software had been performed, we maintained the software through further use of `Specware`™. As we learned more about the problem domain, we made several changes to the requirement specifications. It proved to be easy to re-generate the software from the changed specifications. This was partly due to the fact that none of the changes required

significant redesign. (Eventually, we encountered a change that did.) The initial optimization algorithm used an exhaustive search (for expediency: A discrete optimization expert was not available and our time was limited). The resulting inefficiency led us to encode an additional theory for branch and bound optimization. This theory, unlike many of our earlier, problem-specific ones, is widely reusable. For us, then, maintenance amounts to experimentation with better theories and diagrams. This is not as far from standard software maintenance as it might appear. Industrial developers are always experimenting with improvements and add-ons to their software, the difference being that the experimentation occurs at the software code level instead of in the theory abstractions as in our case.

6 Suggested Improvements

6.1 A Methodology for Understanding Refinement System Interfaces

For maximum ease of use, the user interface of any system must reflect and reinforce the user's mental model of the artifacts and processes involved. The developer of a user interface must keep in mind the education, experience, and general characteristics of the user community. To make the tool usable by a broad range of engineers and/or computer scientists, items requiring an explicit, in-depth knowledge of category theory must be avoided, at least for the current generation of users. It would be best if human factors engineers were called upon to perform a preliminary task analysis for using this technology, including an analysis of domain modeling and requirements elicitation techniques that would be helpful. The emerging understanding of the usability issues with this technology could be used in specifying a user interface and an accompanying methodology for its use.

6.2 Linking Nonformal Requirements to Formal Requirements

To put it bluntly, the people most familiar with application domains may not be interested in authoring requirements in a language based upon higher-order sorted logic and category theory. Natural languages are more widely known and accepted, and provide a sort of freedom in ambiguity of expression with which industrial analysts feel comfortable. The restrictions of rules that seem artificial to their work, however correct these rules might be in the underlying theory, will drive these analysts to seek alternatives to any formal methods that impose such rules. This has been repeatedly observed in our context. In addition, individual communities often have their own notations and/or visual symbology, a phenomenon quite typical of traditional engineering workgroups, not to mention entire disciplines. To make formal or mathematically-based methods of any kind attractive, there must evolve a linkage mechanism to unite the less structured representations of requirements with their formal counterparts. For this, we suggest a contemporary sort of solution: A Web-based interface in which engineers

can "click on" a portion of a natural language document and thereby gain access to formal requirement constructs, usable in a modular fashion. The underlying theory would automate the composition of modules, tracking the user who would be "browsing" the informal requirement document.

6.3 Viewing the Linkage between Requirements, Design, and Software

In the same vein, it would be nice to be able to "click on" formal requirement specifications through the Web-based interface and view the design specifications that reflect those requirements. Specification elements could be viewed individually in this manner—sorts, operations, axioms, and theorems. Parts of the design specifications could be connected to other, related parts in similar fashion, or the design specifications could actually consist of a class of design alternatives, providing the user with a menu of possibilities. The underlying compositonality of the mathematical model would provide the mechanism for putting together elementary specifications for either requirements or designs, again connected through the Web interface. Finally, design specifications could be connected to software fragments or to whole software modules (again, exploiting the compositionality)—"'just a click away".

The theme in all this is visibility and traceability of requirements and designs through the software derivation history. If another analyst were to take over 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? We feel that this is the most important question remaining to be answered.

6.4 Better Derivation Replay and Visibility

Replaying the software derivation for the equipment locator problem requires the execution of 150 operations via the diagrammatic graphical user interface. While this is not difficult, the tool needs to better automate the task by recording the sequence of steps so that they can be replayed simply by "pushing a button". There is structure to these sequences of steps[6]. This structure appears easy to capture, display, and manipulate through the user interface. Note that in making this suggestion, we differ from our previous discussion in the granularity of the refinement constructs being tracked. Here, we suggest tracking objects at the level of specifications and diagrams as opposed to the specification fragments discussed in the previous subsection.

6.5 Improved Specification Libraries

Larger specification libraries available as an initial infrastructure within tools like `Specware`™ would greatly increase productivity. Obvious examples are theories about the constructs encountered in data structure and algorithm courses—trees,

graphs, search algorithms, disjoint set-union-find, and so forth. Among the many others one can name are theories for dimensional analysis and unit conversion of physical quantities. We have experimented with a system of theories about physical quantities known as the Engineering Mathematics Ontology, developed at Stanford[3], and also with a system of theories of pure geometry.

6.6 Better Proof Support

Use of the automated theorem prover currently supplied with Specware™ has forced some limitations upon our use of mechanical theorem-proving. A more widely usable theorem-prover would be a great boon to this technology, as would equation solvers and model checkers, individually or in combination. A more pragmatic approach might be to use a combination of proof checking and automated theorem proving. Why not allow software designers who are used to concocting proofs manually to record their proof steps? Perhaps a proof checker can validate some proof steps fairly easily. Some proof steps are cumbersome, but it might be possible to discharge them interactively. In the worse case, the tool could support the recording of manually-generated but unverified proofs, and these could then be verified by the proof checker manually when convenient.

6.7 Generating State Based Programs

The current version of Specware™ generates only functioning software. No use of state based variables is possible. This limits the efficiency of generated software as well as the use of Specware™ in embedded systems, where improvements to the quality of generated software are highly desirable.

6.8 Software Optimization Transformations

The Kestrel Interactive Software Development system (KIDS)[9] has some very 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 and a simple type of colimit construction (pushouts). The program-level and finite-differencing transformations add to the pushouts the capability to optimize the derived software for efficiency.

6.9 Expressing Rationale behind Design Choices

During design, there are often multiple design alternatives. It is desirable to have a capability to clearly record these alternatives as well as the rationale behind the selection among them that one makes. The rationale could be stated formally, as a trade-off between space and time complexity, for example. When changes in requirements occur over time, this information would be very valuable. All that Specware™ currently stores in a derivation history is a single design alternative. Yet, an initial exploration of a design space yields a great deal of information about the trade-offs available. We propose that that information be captured for future use.

7 Conclusion

We have described our experiences in applying a category-theory-based approach to industrial strength, automated software specification, synthesis, and maintenance. It supports 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, such as mechanical, material, and manufacturing engineering. This suggests the technology provides an approach to general systems engineering that enables one to structure and reuse engineering knowledge in the broadest sense.

References

1. D. Bjorner and C. Jones. *Formal Specification and Software Development*. Prentice-Hall International, 1982.
2. L Blaine and A Goldberg. Dtre - a semi-automatic transformation system. In B Moller, editor, *Constructing Programs from Specifications*. North-Holland, 1991.
3. T. Gruber et al. An ontology for engineering mathematics. In *Proceedings of the Fourth International Conference on Principles of Representation and Reasoning*. Morgan Kauffman, 1994.
4. J. A. Goguen and R. M. Burstall. Some fundamental algebraic tools for the semantics of computation–part 1: Comma categories, colimits, signatures and theories. *Theoretical Computer Science*, 31(1,2):175–209, 1984.
5. J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146, 1992.
6. R. Jullig and Y. V. Srinivas. Diagrams for software synthesis. In *Proceedings of KBSE '93: The Eighth Knowledge-Based Software Engineering Conference*, pages 10–19. IEEE Computer Society Press, 1993.
7. J. Meseguer. General logics. In *Logic Colloquium '87*, pages 275–329. Science Publishers B. V. (North-Holland), 1987.
8. B C Pierce. *Basic Category Theory for Computer Scientists*. MIT Press, 1991.
9. D. Smith. Kids: A knowledge based software development system. In M. Lowry and R. McCartney, editors, *Automating Software Design*. MIT Press, 1991.
10. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1992.
11. Y. V. Srinivas and R. Jullig. `specware`^{tm}: Formal support for composing software. In *Proceedings of the Conference of Mathematics of Program Construction*, 1995.

12. T. C. Wang and A. Goldberg. A mechanical verifier for supporting the design of reliable reactive systems. In *Proceedings of the International Symposium on Software Reliability Engineering*, 1991.
13. K. Williamson and M. Healy. Formally specifyuing engineering design rationale. In *Proceedings of the Automated Software Engineering Conference-1997*, 1997.
14. K. Williamson and M. Healy. Deriving engineering software from requirements. *Journal of Intelligent Manufacturing*, 11(1):3–28, 2000.