Systems Synthesis: Towards a new paradigm and discipline for knowledge, software, and system development and maintenance

Keith Williamson
Mathematics and Computing Technology
Boeing Phantom Works

March 15th, 2001

This paper motivates and describes a new paradigm and discipline for knowledge, software, and system development and maintenance. This paradigm promises to improve system quality and make systems development and maintenance faster and cheaper.

Table of Contents

Figures

Chapter 1: Executive Summary

This paper motivates and describes an emerging and enabling technology for systems development. This technology solves several fundamental and unsolved challenges for software and systems development and maintenance. We hope to develop a program aimed at a phased maturation of this technology. This, in turn, is aimed towards eventual technology transfer of this technology into industrial software and systems development..

Chapter 2: Challenges

Government agencies, corporations, and businesses are increasingly automating their work processes by developing computer systems that assist people with their work. However, the current state of software engineering and software development technology presents three major challenges; software systems:

1. Are expensive and time consuming to develop,
2. Do not always perform what users want or need,
3. Are difficult to maintain and adapt to changes in requirements.

Systems development and maintenance needs to be made faster, cheaper, and better. These are the central challenges addressed by the research and technology development program laid out in this paper. In this chapter, we will discuss these three challenges in reverse order.

Section A: *Software Systems Adaptation*

Let us begin our exploration of these issues by first focusing on the challenge of maintaining and adapting software systems. Companies such as Boeing are increasingly embedding corporate knowledge, engineering design rationale, and human expertise into software systems. The following list demonstrates the pervasiveness of engineering software systems that have been developed at Boeing:

- Numerical Analysis – math and statistics library, optimization software.
- Preliminary Design – wing and airflow analysis, airplane configuration.
- Systems Design – Tube Routing such as Genesys, Kirts, Super Router.
- Structures Design – Various ICAD applications; e.g., wing box design.
- Detailed Part Design – ICAD systems for design of shear ties, stringer clips, etc.
- Materials Engineering – ESDS (engineering standards distribution system).
- Tooling – NC Tooling, ICAD-based design of tools (e.g., lay-up mandrels).
- Manufacturing – Factory scheduling, automated process planning.
- Electrical – Electrical connectors selection, LRU equipment locator.
- Avionics – Mission planning, flight control, navigation, sensor fusion.
- Propulsion, System Communication, Human Factors…
- And broadly, systems engineering and integration

Many of these systems were designed for very specific contexts, or requirements. When requirements change, software systems must change accordingly. Experience suggests this is often a costly and time-consuming process . In addition to evolving requirements for specific tasks, we would like to leverage the knowledge embedded in these systems for other, related tasks (e.g., using knowledge about a manufacturing process in both a design and manufacturing context). However, this goal has been hard to achieve as well. The reasons for these problems are complex, but a large challenge arises from the attempt to reuse knowledge at the software level.

Subsection i: The Challenge of Software Reuse

Software is the end artifact of a long and complicated process that goes from user requirements, through a process of design, to an implementation, which is built on top of some virtual machine ([ICSR2000]). 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. Even if the constraints and assumptions are understood, and fit current needs, they may not fit a future need. If the original requirements and design rationale have not been made explicit, it is difficult to adapt the software to meet this future need.

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. Instead of simply striving for software reuse, we would like to record the intellectual effort that goes into software systems development and leverage that for other purposes.

However, accomplishing this with current software development technology is difficult due to multiple representations for various software artifacts (e.g., different languages for requirements, architectures, designs, and programming), the inability to seamlessly, and effectively, interoperate between these representations, and the inability to easily trace and verify requirements through the design process to software.

Subsection ii: Knowledge Management

Consider the following statistics from the United States Bureau of Labor Statistics (taken from http://stats.bls.gov/news.release/tenure.t06.htm):

Median years of tenure with current employer for employed wage and salary workers by occupation

	1983	1987	1991	1996	1998	2000
Engineers	6.3	6.1	6.7	6.6	5.3	4.8
Mathematical and computer scientists	3.8	5.0	4.2	4.5	3.3	3.3

Institutional memory is being lost in industry. The knowledge of how computing systems are developed and why is increasingly being lost due to high labor turnover rates. Yet this knowledge, if made more explicit, could be a key asset for an institution or corporation such as Boeing. Hopefully this knowledge can be codified, structured, used, reused, and evolved, in an easy to use, but disciplined, manner. This calls for a corporate knowledge management system.

There are many criteria for determining whether and how knowledge should be stored and explicitly represented, including ([AnnieBrooking]):

1. Longevity of knowledge
2. Extent of knowledge growth and change
3. Return on investment for codifying knowledge

4. Size of problem space of knowledge applicability
5. Complexity of problem knowledge addresses

Let us assume that it is worth codifying software development knowledge for some computing systems. It is important to recognize that any knowledge management system exists in the context of a corporate culture, management philosophy, and management and business processes. While technology can enable people to capture, share, and access knowledge more easily, it cannot make people contribute and share knowledge. Accompanying changes in corporate culture and management practices are needed (e.g., new incentive systems to encourage longer term planning). While these issues are crucial, the focus of this paper is upon enabling technology.

Section B: *User-Centered Computing*

Let us turn now to our second major challenge – building computing systems that accomplish what users want and need. A report by the Standish group (see http://www.standishgroup.com/visitor/chaos.htm) describes why software development projects fail:

- Successful Projects
  - Good User Involvement
  - Executive Management Support
  - Clear Statement of Requirements
- Projects that either were Over Budget, or delivered Incomplete Functionality
  - Lack of User Input
  - Incomplete Requirements & Specifications
  - Changing Requirements & Specifications
- Canceled Projects
  - Incomplete Requirements
  - Lack of User Involvement
  - Lack of Resources

Clearly, a major challenge for software development is more effective involvement by end-users in the elicitation of system requirements.

Methods for eliciting requirements from end-users can range from interviews, to questionnaires, to task analysis (ideally by human factors engineers) within the context of actual work scenarios. Since some forms of knowledge are often tacit, it is important to include some form of contextual task analysis . By contrast, common approaches to requirements elicitation involve primarily interviews and requirement reviews (which are not done within the context of actual work environments). After a fairly complete working version of the system has been developed, usability testing and analysis is sometimes performed. Of course, any changes that are uncovered at this point in time are often difficult to implement (for reasons discussed in the previous section). Computing systems need to be made in such a way that usability and adaptability by end-users is fundamentally built into them.

A rapid prototyping approach to system definition and development can make requirements elicitation iterative. This approach initially provides user interface

prototypes with limited functionality, in a manner akin to storyboard use in movie script development. Functionality is added to the evolving system to meet evolving requirements. Continuing cycles serve to refine the system definition and development. Combining this approach with contextual task analysis makes it possible to elicit a more complete and correct set of end-user requirements. Thus, we are led to a user-centered, adaptive approach to system definition and development.

Section C: *Software Development*

Let us now consider the first major challenge we mentioned at the beginning of this chapter - software systems are expensive and time consuming to develop. In addition to the problems already discussed in this chapter, software development is hindered by two major factors – a general lack of correctness in the software development process, and the shear size and complexity of modern computing systems.

Even if one assumes that system requirements have been adequately captured, the lack of an ability to assure correctness (that the software does what is needed) in the software development process leads to high software testing costs. Software testing can consume as much time as (or more than!) software development. While testing will never be entirely eliminated, it can be greatly reduced with "correct by construction" software synthesis technology. Synthesized software can often be proven to satisfy requirements (as will be seen below).

Perhaps the greatest challenge of all lies in the shear size and complexity of modern computing systems. Large computing systems involve the integration of multiple software and hardware components. Overall system functionality, or requirements, must be driven down (or built on top of) the functionality provided by component systems. Each component may have its own inherent complexity. Descriptions of component interfaces are often informal, ambiguous, and not at the right level of abstraction. This leads to great challenges in designing systems and assuring their overall correctness. Systems, or integration, testing is a separate area of testing geared towards empirically evaluating the correctness of a system of components. This branch of testing is an extremely costly and time-consuming process. If one were responsible for integrating a COTS (commercial off-the-shelf) payroll system with a COTS employee benefit system, one would understand these issues better! Or better yet, integrating various avionics systems to achieve flight control and management.

A computing system must be understood and designed at appropriate levels of abstraction. Too much unnecessary detail introduced early in the design process leads to systems that are unnecessarily complex, inflexible, and faulty. Abstraction, modularity, and loosely coupled systems, can be used to help manage complexity. Object-oriented systems certainly help software development in this regard. However, OO methodologies lack abstraction capabilities that are necessary for complexity reduction. Nevertheless, managing complexity of large systems is a daunting task! What appears to be lacking in the current state of software development and maintenance technology is having an infrastructure that is fundamentally oriented towards managing complexity at all levels. As we shall see, a branch of mathematics, category theory (which concerns itself with structure and structure preserving mappings), provides a foundational theory that allows knowledge to be expressed at different levels of abstraction, and uses the structure of this knowledge to help manage system complexity.

The development and maintenance of computing systems needs to be made faster, cheaper, and better. But as we have seen, the current state of software development technology must overcome the following challenges to enable this to happen:

- Difficulty of leveraging knowledge embedded in software for:
  - Evolving a task, or
  - Adding related tasks.
- Multiple representations of various software artifacts.
- Lack of requirements tracability in software construction.
- Lack of user-centered, adaptive approach to system definition and development.
- Lack of correctness in software construction.
- Shear complexity of modern computing systems.

Chapter 3: Software Synthesis Technology

This chapter describes a new paradigm and discipline for software system development and maintenance. This paradigm addresses the challenges laid out in the previous chapter. To some extent, this technology is currently available and has been used and evaluated in industrial settings. However, further evaluation and maturation of this technology is needed.

Section A: *General Background*

The notions of software generation and software synthesis have been around for some time in various guises. In general, the goal is to help automate the construction of software systems from a description of requirements. However, most of these tools either do not generate complete software solutions or do not generate general-purpose software.

There are commercial tools (e.g.; see http://www.rational.com/) that allow generation of software templates based on diagrams written in the Unified Modeling Language (UML, which is a standard design language [UMLdistilled]). Software templates capture the syntax of data types and their operators, but are limited in their expressiveness of the underlying meaning, or behavioral semantics, of data and operations. While UML contains the Object Constraint Language (OCL), which does have the ability of describe the semantics of operations (through the use of invariants, pre-conditions, and post-conditions), current commercial tools do not generate software based on OCL descriptions. Without this capability, the correctness of the final software solution is hard to determine without a good deal of testing.

There are languages and tools for generating software from finite state machine representations of requirements (e.g., the tools available from http://www.ilogix.com/, which are also based on UML). Often geared towards embedded software, these tools are able to generate complete software systems that implement the requirements as stated in finite state machine representations. However, the expressiveness and computational capability of finite state machines is limited, which prevents their use for general-purpose software description and generation.

Section B: *Software Refinement Systems*

There are a few languages and tools that support the generation of complete and general-purpose software systems from requirements. Both the Vienna Development Method (VDM) and the B-Method have languages and tools that support a refinement-based approach to software generation [VDM, B-Method]. One starts with a requirements statement in a general-purpose algebraic specification language, and gradually and iteratively refines this into successively more detailed specifications, eventually arriving at a specification that maps directly into some virtual machine (e.g., the programming languages C, Ada, or Lisp). Each refinement step involves proving, or demonstrating, that the semantic properties of a requirements specification are upheld in the refined specification (automated theorem provers are provided to assist with these proofs). Software could, in principle, be proven to meet requirements. Requirements traceability can be provided based on these proofs. These methods have been used in Europe for developing software systems such as a metropolitan railway management

system [VDMcitation] and a manufacturing production cell control program [Bcasestudies]. These methods adequately meet all of our goals except two – managing complexity and user-centered design.

Like VDM and the B-Method, Specware™ supports the generation of complete and general-purpose software systems from requirements [Specware]. Unlike those methods however, Specware™ is based on category theory. Specware™ generalizes the notion of how a specification can be embedded in another specification, and uses diagrams and colimits (from category theory) as a more general mechanism for composing specifications (see Figure 1 and 2). Specification refinements can be composed using similar constructs from category theory. These more general compositional mechanisms improve the management of system complexity [Specware].

Section C: *Specware™*

The paradigm embodied in Specware™ is one that allows for the capture and structuring of formal requirement specifications, design specifications, implementation software, and the refinement processes that lead from requirements to software ([JASE2001]). In this approach, the refinement process can guarantee correctness of the derived software. By recording, replaying, and modifying the software derivation history, we are able to more easily maintain the software. By capturing, abstracting, and structuring requirement, design, and implementation knowledge, 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.

Specware™ consists of a specification language, specification refinements, constructors from category theory, an underlying theorem prover (tied to the specification language), and mappings from subsets of the specification language to target programming languages (e.g., Lisp and C). The next two pages, and figures, will be somewhat technical, but I ask for the readers' diligence and attention. What you will hopefully see is a general-purpose infrastructure that corresponds fairly naturally to the manner in which programmers go about doing their work, though much of this work is currently rarely documented in an explicit and complete fashion.

Subsection i: Specifications, Morphisms, and Diagrams

Primitive components of Specware™ are signatures, signature morphisms, specifications, and specification morphisms (see Figure 3). *Signatures* capture the syntax – the terms used to express data types and their operations. *Signature morphisms* are mappings between signatures that embed terminology of one specification into another specification. *Specifications* build on signatures by adding rules describing the semantics (axiomatic statements of properties holding between terms). *Specification morphisms* are signature morphisms that preserve these properties (modulo signature morphism). A theorem prover can be used to assure this. Specifications are components of knowledge, and can be used to describe portions of requirements, architectures, designs, or implementations. Specification morphisms express the structural relationship of embedded knowledge between specifications.

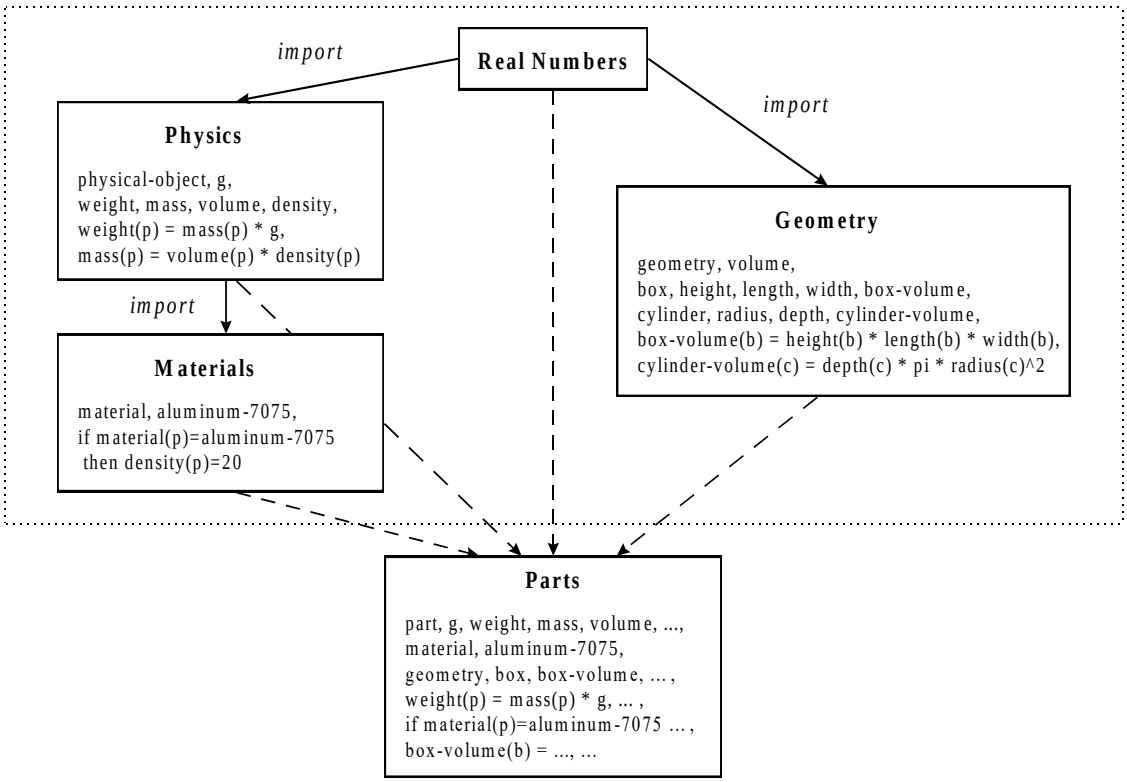


Figure 1: Construction of Parts Requirement Specification

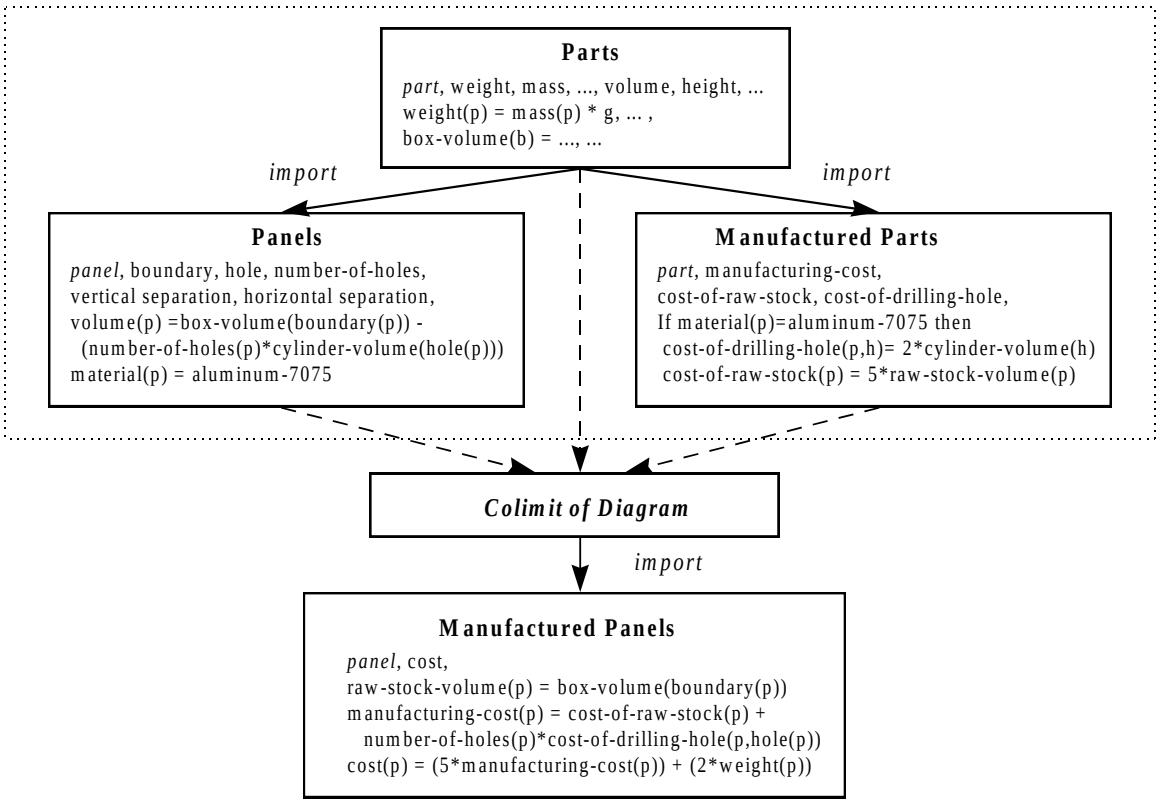
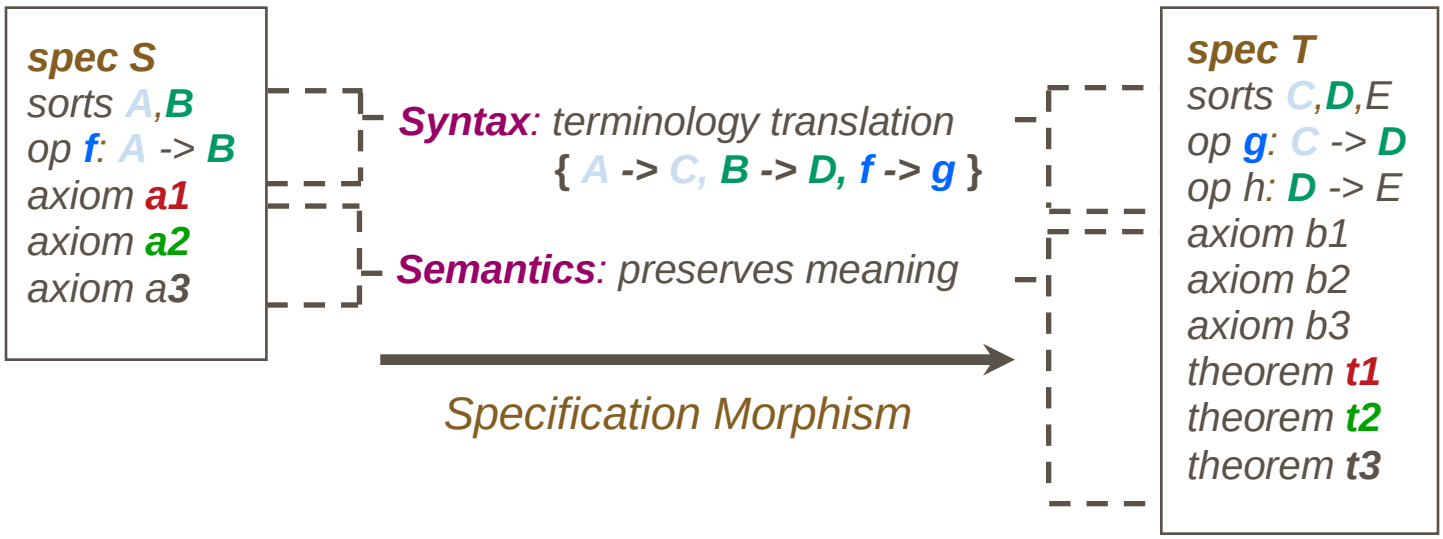


Figure 2: Construction of Manufactured-Panels Requirement Specification



- Syntax:
  - Sorts in **S** map to Sorts in **T**
  - Ops in **S** map to Ops in **T**
  - ***Compatible mapping*** of Ops in **S** to Ops in **T**
- Semantics:
  - Axioms in S must translate to Theorems in **T**

Figure 3: Specifications and their Morphisms

Collections of structurally related specifications can be described in *specification diagrams* (the dashed rectangles in Figures 1 and 2). The *colimit* of a specification diagram is a specification that captures the overall system specification described in the diagram[1]. Colimit specifications can be automatically generated in Specware™. The specifications outside the diagrams in Figures 1 and 2 are colimits of their respective diagrams. Specification diagrams and colimits allow us to represent and decompose knowledge at an appropriate level of abstraction. Determining the appropriate level of abstraction is left to the specification writer (e.g., requirements analyst or designer), and may evolve over time.

Subsection ii: Specification Refinement

By writing specifications and using diagrams and colimits, one can construct a specification describing system requirements. To synthesize software, interpretations are used to express the refinement of a specification (see Figure 4). At all levels of systems development, one is always mapping from some requirement specification to some virtual machine specification by writing a specification that satisfies requirements and is operationally defined in terms of the virtual machine. When all the requirements are refined to a point where they can be mapped directly to a concrete virtual machine (e.g., some programming language), software can be generated that satisfies the requirements.

A *definitional extension* of a specification B is a specification C that adds only constructive definitions to B[2]. This is the heart of programming. Whenever one writes part of a program, one adds data types and operations that are operationally defined in terms of data types and operations provided by some virtual machine. An *interpretation* from specification A to specification B is a specification morphism from A into a definitional extension C of B. The morphism from B to C shows what the virtual machine is, and how the program builds on top of it. The morphism from A to C is the morphism that assures requirement properties are preserved. This morphism makes explicit the proof obligations that are necessary to demonstrate that software does what it should. All this may seem technical, but in essence, this is a large part of what programmers implicitly accomplish when they program.

Subsection iii: Refinement Composition

An interpretation, or implementation, can be constructed automatically from other interpretations in Specware via two general mechanisms – sequential and parallel refinement. In *sequential refinement*, two or more interpretations can be composed in sequence. In Figure 5, we see that sets can be implemented in terms of bags, and bags can be implemented in terms of lists. Thus, sets can be implemented in terms of lists. This corresponds to an iterative style of program development. Note that in Figure 5, a ”cross logic” morphism is alluded to. These morphisms go from the logic of the specification language to the logic of a programming language. More will be said about this in the next chapter.

---

[1] Technically, the colimit specification is actually the apical object of the colimit cocone.

[2] Syntactic restrictions in definitions enable translation to a target programming language.

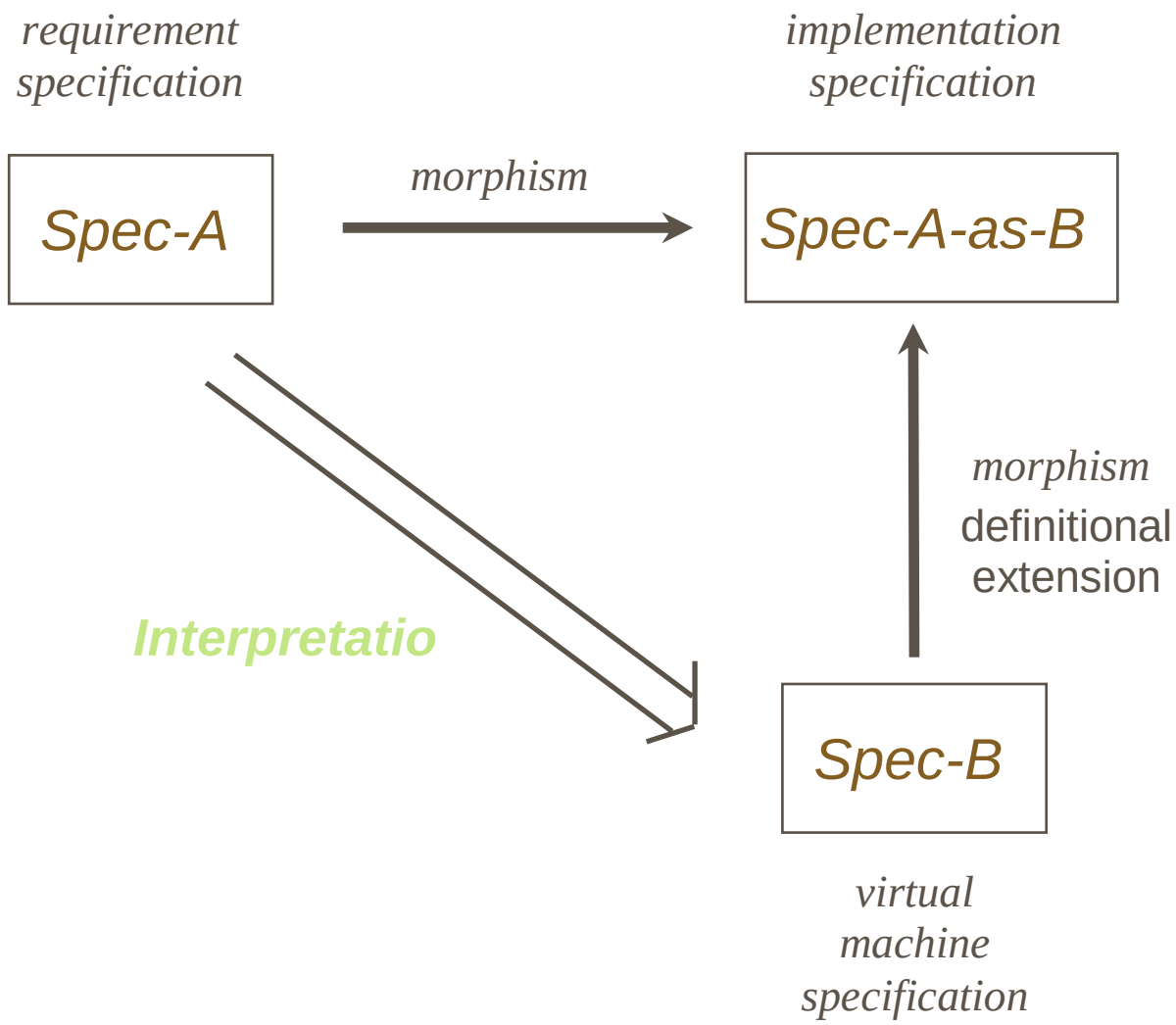


Figure 4: Interpretations

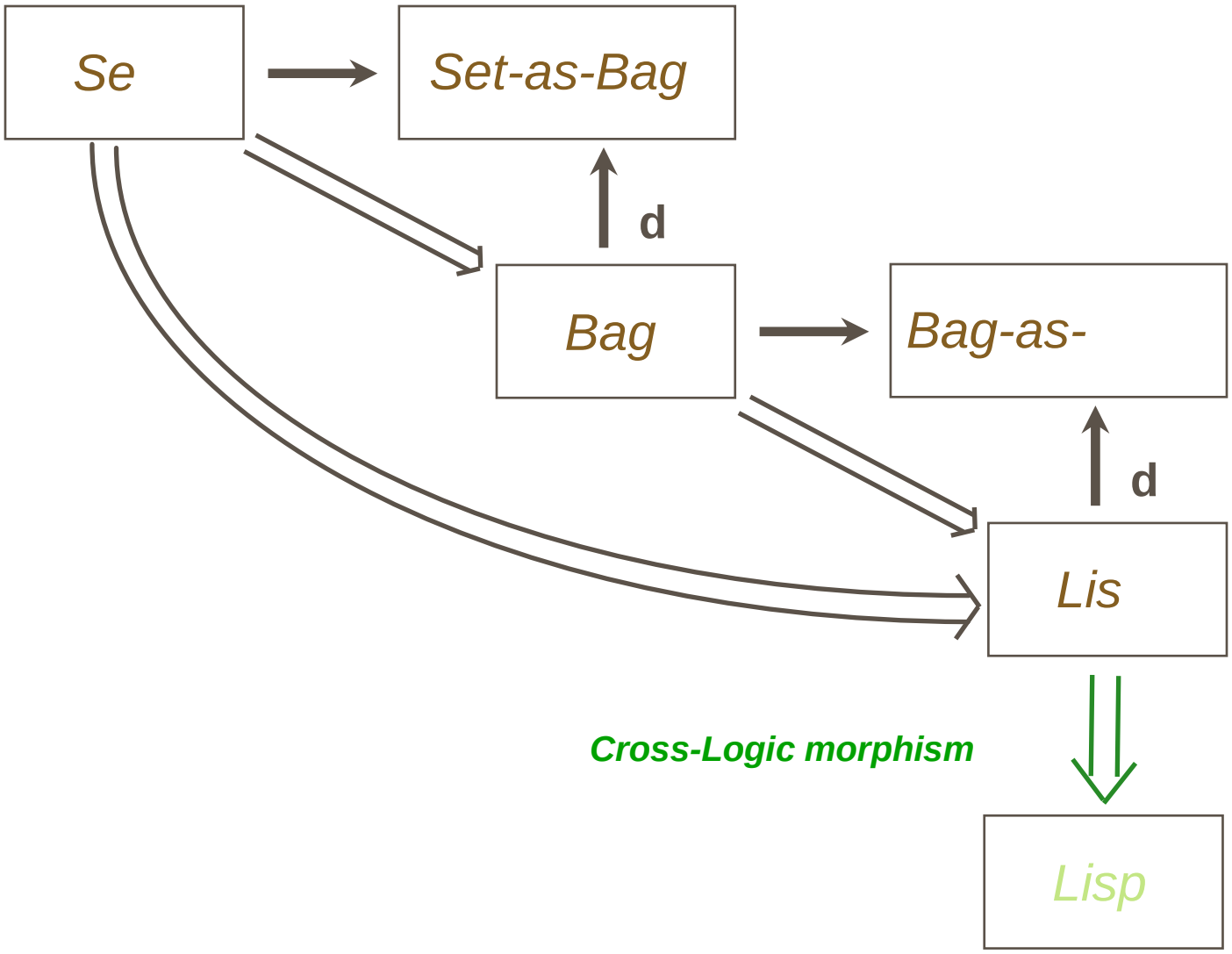


Figure 5: Sequential Composition of Interpretations

The other mechanism for constructing interpretations is via *parallel refinement*. Given a diagram of specifications and its colimit, and interpretations for each of the specifications in the diagram[3], an interpretation for the colimit specification can be automatically constructed. In Figure 6, we have a colimit specification that is a statement of an optimization problem for the layout of manufactured panels. This colimit specification is constructed from a diagram of component specifications. Each of these specifications has parallel interpretations. From this information, an interpretation for the entire panel layout problem can be automatically constructed. In this constructed interpretation, the "implementation" specification is a consistent amalgamation of the constituent implementation specifications, and the "virtual machine" specification is likewise an amalgamation of the constituent virtual machine specifications.

One can view parallel refinement as a general mechanism for program construction from components. This general ability to decompose the programming task into subtasks brings manageability to the construction of complex systems. Note that no matter how interpretations are arrived at, one is always going from requirements to some virtual machine. If the requirements are complex, the refinement process decomposes based on the structure of the knowledge (specifications, in this case). The overall decomposition and design of a system might look something like Figure 7. In this sense, category theory provides a foundational theory that allows knowledge to be expressed at different levels of abstraction, and uses the structure of this knowledge to help manage system complexity.

Subsection iv: Software Synthesis in Specware™

A person can derive software with Specware™, by writing requirement specifications, assembling them into larger specifications, writing interpretations, and assembling them into larger pieces (through sequential or parallel refinement). Eventually all requirement specifications get decomposed as much as needed, and eventually every interpretation goes to a concrete virtual machine. The creative aspects of programming are left to the programmer (deciding how to decompose and how to implement); the tedious aspects (assembling components in a consistent fashion and checking that requirement properties are upheld in implementations) are left to Specware™. In this semi-automated approach to software development, we have documented and structured requirements, architectures, designs, implementations and the refinements processes (and proof obligations) that link them together.

[3] Additionally, there needs to be consistency across interpretations that jointly implement shared functionality (see [Specware]).

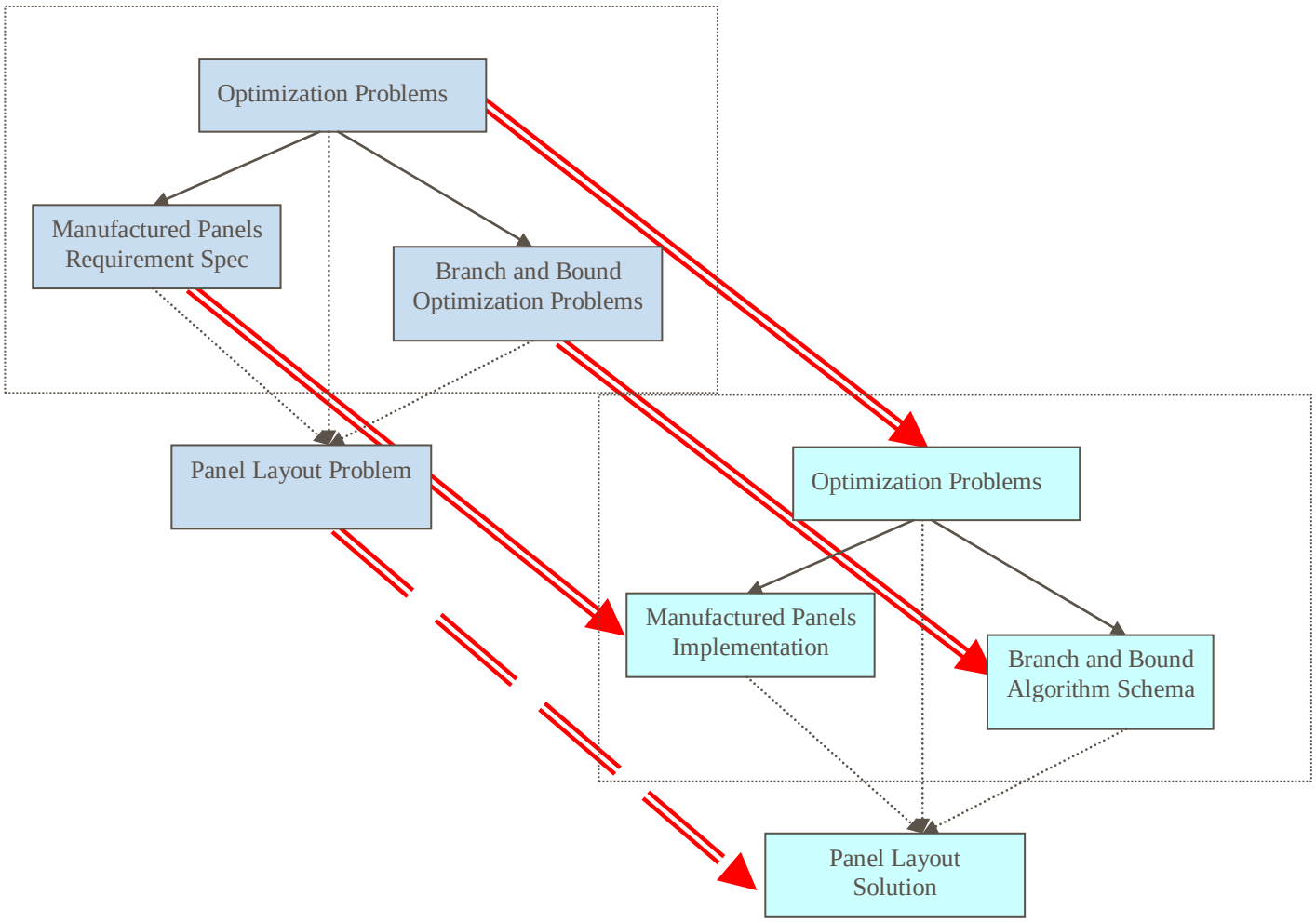


Figure 6: Parallel Composition of Interpretations

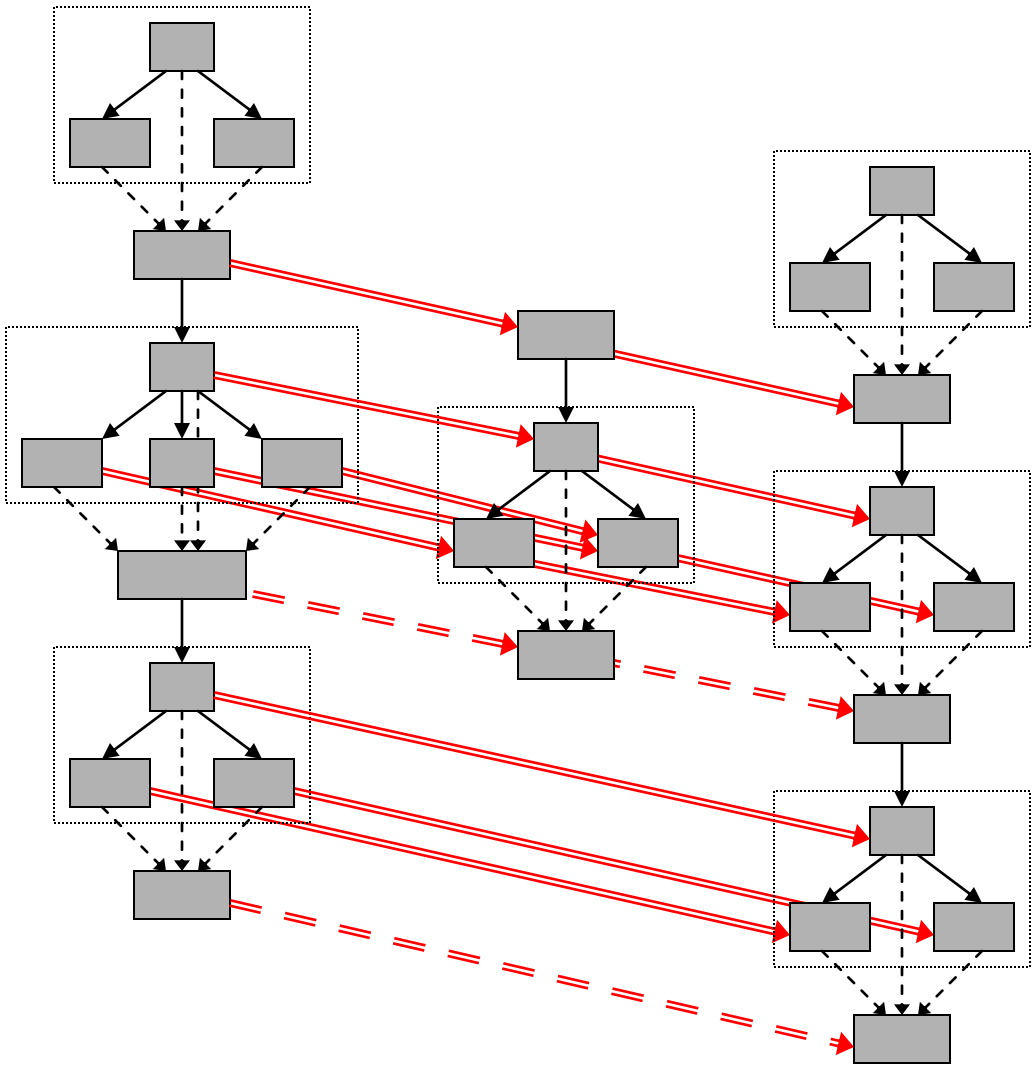


Figure 7: Refinement Composition to Capture Design

While the quality of derived, or synthesized, software can be assured through the use of automated theorem provers (and other mechanisms such as constraint solvers), maintenance costs and subsequent development times are reduced through the ability to automatically replay software derivations in response to requirement modifications. A change to a requirement specification can be made, and an attempt can be made to apply the prior software derivation. If a theorem prover is being used to check that requirements are being upheld, any potential problems with applying the previous designs will be pointed out automatically. Even if some of the previous designs need to be reworked, presumably many parts of the old design still apply. Specware™ applies what it can of the previous derivation, and points out design rework. Of course, this assumes that theorem proving capabilities are adequate for the task at hand.

What Specware™ enables is:

1. The explicit capture and structuring of the knowledge that goes into the definition, design, and development of software systems,
2. The partial automation of using this knowledge for developing and maintaining software systems that provably satisfy requirements,

By making this institutional knowledge explicit, and using it in a semi-automated process for software synthesis, this knowledge of system definition, design, and software derivation can become a key institutional or corporate asset.

Section D: *Specware™ Use*

Specware™ was developed at Kestrel Institute, in Palo Alto, CA. It has been applied there to several areas, most notably in the area of high-performance schedulers that have been applied to scheduling and planning of military logistics [Smith]. But it has also been applied to such things as developing a provably correct implementation of a part of the Java Virtual Machine bytecode verifier [Coglio]. In the scheduling arena, there has been considerable effort spent to formalize the knowledge of algorithms and data structures associated with various classes of scheduling problems. The interface to Specware™ has been tailored to the domain of scheduling problems, shielding the end-user (to a large extent) from having to know much about logic and algebra (and nothing about category theory).

Peter White at Motorola applied Specware™ to a portion of the design of secure operating system kernel [Motorola]. While software was not generated in this case, the requirement specifications and some of the design specifications were written and developed with Specware™. This work did lead to a developed product.

At Boeing, we have applied Specware™ to synthesizing software that supports more traditional engineering disciplines. In 1997, we generated requirement specifications, design specifications, and implementation software for a small component that was part of a mechanical engineering design task [JIM]. In 1998 and 1999, we did the same thing, but this time we tackled a much larger engineering design application, the equipment locator problem [ASE99]. This is a problem of finding optimal layouts of electrical equipment to positions on shelves. Not only did we generate this application along with the proofs (often done manually) that demonstrated the software was correct,

but also in 1999, we were able to make changes to various specifications and replay the software derivation (thus maintaining the software with Specware™).

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 for 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 ([JIM2000]).

Section E: *Specware™ Evaluation and Maturation*

The National Security Agency (NSA) recently sponsored an evaluation of two potential methodologies for producing secure software systems [Widmaier, et al.]. One was CMM Level 4, represented by a contractor team at "Company L". The other was Specware™, represented by a contractor team at "Company M". Both are large aerospace companies whose names were not revealed. The University of Maryland was hired to perform the evaluation. The NSA supplied a requirements document (an informal specification) for a card-key entry system. The document had some built-in ambiguities, left there to make the exercise realistic and to see what affect these would have on the two methodologies. Both teams were given five months before turning their results over to the University of Maryland evaluators.

The CMM team was done in two months. The Specware™ team took four. This is the same phenomenon noticed at Boeing, where it takes about twice as long to develop the full set of formal specifications up front, along with the morphisms, refinements, and diagrams for category-theoretic synthesis. The results were quite interesting. When evaluated, the CMM-developed system had a 56% success rate. 56% of the time, a user either gained entry to a building when they were supposed to or was denied entry when not. On the other hand, the Specware-developed system had a 77% success rate. But the story doesn't end there.

There was an ambiguity in the requirements document that could be interpreted in one of two ways. The CMM team happened to interpret it the right way, but the Specware™ team got it wrong, leading to a large negative impact. It was determined that a simple change could be made at an abstract level in the software derivation, and that new software could be synthesized rather quickly. This change would increase the system success rate to 98%. In looking at the CMM developed system, it was determined that the needed changes were dispersed and unclearly documented. At most a few percentage points' improvement could be made without a thorough, and costly, rework of the (relatively undocumented) system development.

The overall conclusion of the NSA study was that the Specware approach was better by far, but that the technology was not sufficiently mature for wide application. More specific recommendations for the continuing evaluation and maturation of Specware™ were made in [JASE]. These can be broken into two broad categories. The first is a set of unknowns, which present risks for eventual wide-scale use of this technology. These areas need further evaluation:

1. Effectiveness of general theorem proving capabilities
2. Can software synthesis scale to large-scale applications

3. Can efficient embedded (i.e., state-based) software be generated
4. To what extent are costs and times saved when reusing knowledge:
    a. Need to carefully examine maintenance costs for single system
    b. Need to carefully examine saving across sets of related systems

Secondly, areas of maturation in the technology include:

1. Improved coverage and usability of tools
2. Derivation capability for user-interface software
3. Rapid prototyping capability added
4. Larger specification library infrastructure

Of all of the challenges outlined at the end of Chapter 2, only a user-centered, adaptive approach to system development is missing in Specware™. Actually, the adaptive part is there, but the ability to have this adaptation driven by the end-user is missing. This is what the first three items in this list address.

Chapter 4: Systems Synthesis

Before getting to specific proposals aimed at further evaluation and maturation of Specware™, it is worth discussing the underlying theory in a bit more depth. Category theory [MacLane, Pierce, Crole] is an abstract branch of mathematics that is increasingly being applied in more concrete settings (e.g., computer science and physics). In this chapter, we describe how category theory can play a foundational role in the definition, development, and maintenance of complex systems [GoguenSystems]. What we will urge is that foundational research accompany the evaluation and maturation of Specware™.

By broadening the scope of research and technology to be matured, we place ourselves in a better position to handle *heterogeneous systems* - systems composed of different types of hardware and software subsystems. As will be seen, these subsystems can embody disparate computational paradigms (e.g., a Java-based component linked to a finite-state-machine component linked to a neural-network component). In principle, category theory, in its application to general systems theory, can be applied to systems as complex as flight control systems (including the physical design of flight control surfaces and their supporting hardware and software subsystems). This has yet to be demonstrated in an industrial setting, however.

Section A: *Category Theory*

A *category* consists of *objects*, *arrows* (or *morphisms*) between objects, and a *composition operation* on pairs of arrows (that can be linked) that is transitive[4]. A *diagram* in a category is a collection of its objects and arrows. Roughly speaking, the *limit* of a diagram identifies the shared portions in the diagram, while a *colimit* identifies the synthesis of objects in the diagram. In many categories, it can be shown that limits and colimits always exist for any diagram. In fact, the proofs of these properties often lead to constructive ways of automatically computing limits and colimits.

A *functor* between two categories maps objects to objects, and arrows to arrows, in such a way that the structural relationships holding in the source category still hold in the target category (e.g., mapped arrows go between appropriately mapped objects, and compositions are preserved by the mapping). A functor can map multiple objects (or arrows) to a single object (or arrow). Functors are structure-preserving mappings between categories. They show how one category can be embedded in another (see Figure 8 for a general example of this).

It is often useful to think of a functor from A to B as a picture of A embedded in B. Given two functors F and G between categories A and B, a *natural transformation* is, roughly speaking, a way of mapping, or translating, F's picture (of A in B) to G's picture (of A in B). See Figure 9 for an example that will be discussed later in this chapter.

[4] This is a simplification, as it ignores identity morphisms and how they compose with other morphisms.

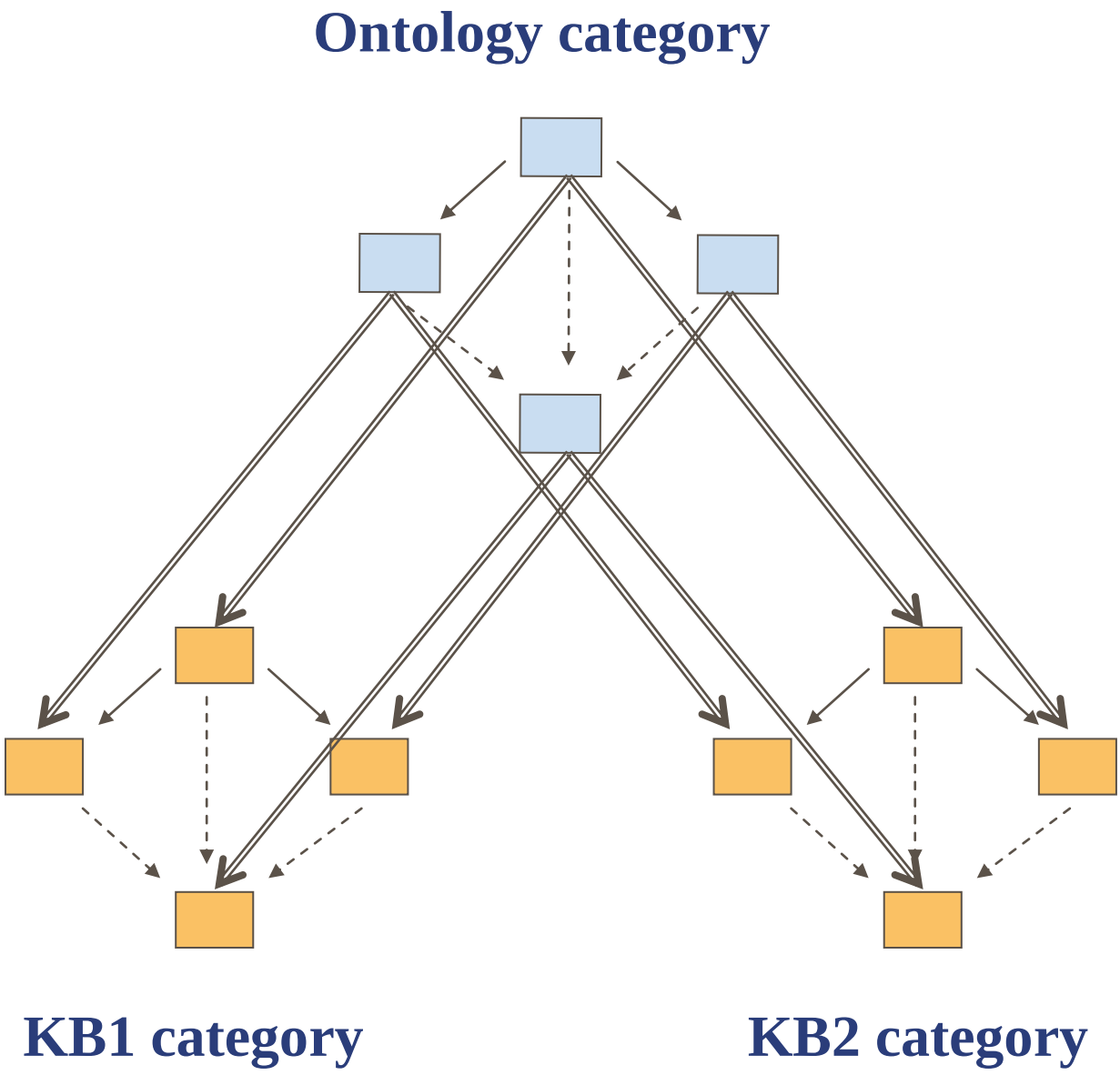


Figure 8: Functors Manifesting Shared Meaning

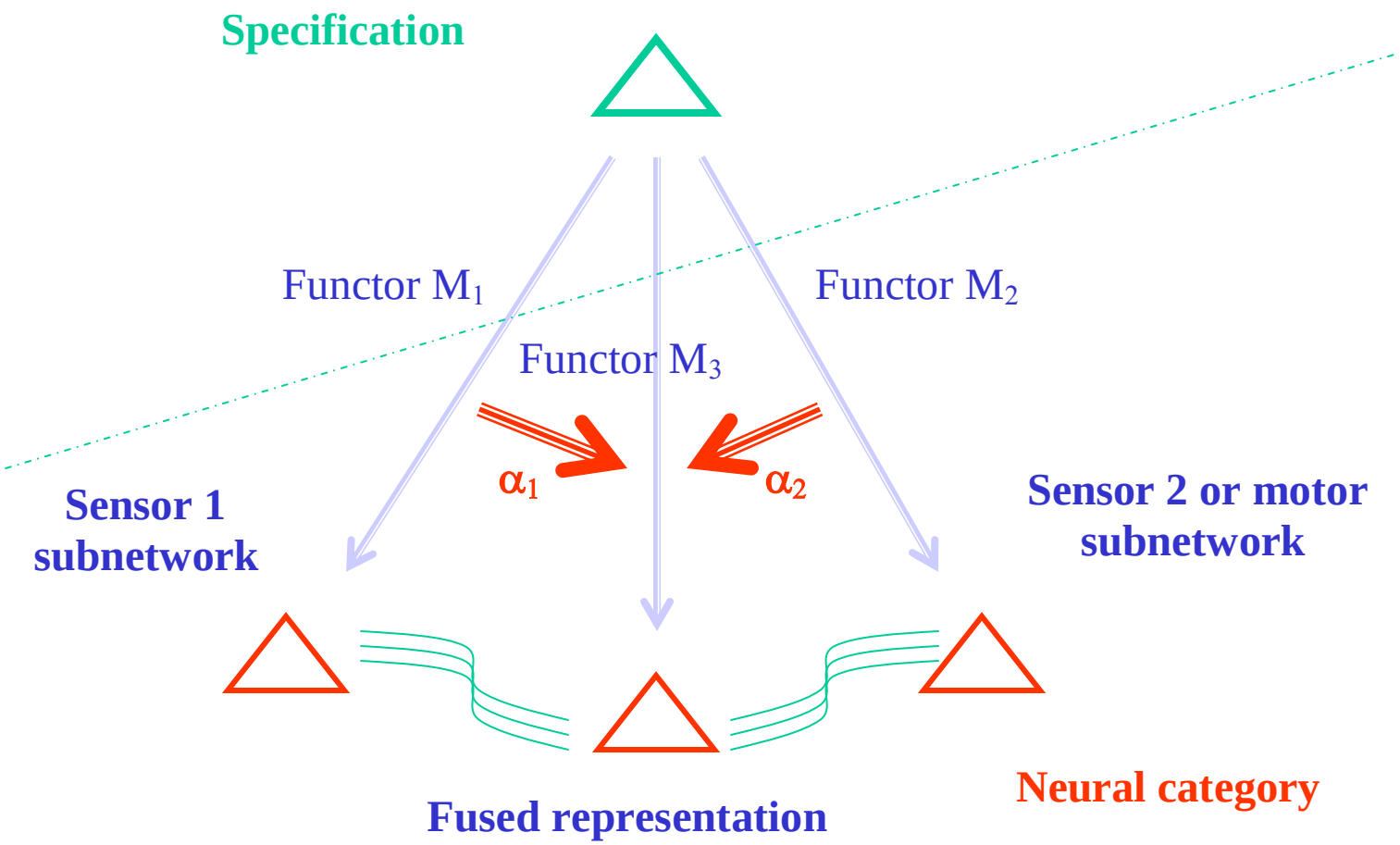


Figure 9: Natural Transformations (Red Arrows) Fuse Representations

In Specware™, there are multiple categories underlying the tool. To name a few important ones, there are categories for specifications, interpretations, and shapes (parallel refinement does not always map from a diagram into another diagram having the same shape as the first). To a large extent, the benefit of category theory lies in its general applicability. In each of these categories underlying Specware™, a diagram can be formed, and colimits can be automatically constructed. This general mechanism works for each of the categories underlying Specware™.

Section B: *Logics and their Semantics*

We mentioned cross-logics morphisms in the preceding chapter. In going from the underlying logic of a specification language to the underlying logic of a programming language, there are two categories - one for the specification language, and one for the programming language. In the terminology of category theory, there is a functor, a structure-preserving mapping, that goes from one category to the other. As with the interconnectedness of word definitions in a dictionary, what is important is the structural connections between objects and preserving that structure when mappings are involved. Arguably, elevating structure to be as important as the objects themselves is the idea that distinguishes category theory from set theory as a foundation for mathematics, and in our case, as a foundation for systems development.

In many complex systems, subsystems may embody different computational paradigms. There are different abstract models of machines and computation [Denning]. These include machines like Finite State Machines, Pushdown Automata, and Turing Machines. Other abstract models of computation include Petri Nets, Lambda Calculus, Logic Programming, and Neural Networks. In general, each of these machines has an associated language and logic. In heterogeneous systems, we must map between and among these machines, languages, and logics in such a way that overall system coherence is achieved. We want them to work together correctly and seamlessly.

The theory of general logics [GeneralLogics] and institutions [Institutions] provides a general theory for ensuring that mappings preserve meaning in heterogeneous systems. For each logic, there is a category in which the objects correspond to statements in that logic, and the morphisms correspond to proof (or computation) in that logic. There is another category in which the objects correspond to the models (the semantics of the logic), and the morphisms correspond to mappings between models. There is a functor going from the syntactic category to the semantic category, and an adjoint (or reverse) functor mapping from semantics to syntax. These are the ideas behind the theory of institutions. In the theory of general logics, we are able to map between logics in such a way that provability (or computation) is preserved. Providing functors that map between the syntactic categories of each logic achieves this. Of course, we want to do this is a way that preserves the underlying semantics of each logic. Thus, the theory of general logics is tied to the theory of institutions.

What we would like is the ability to describe a system that is composed of subsystems. Each of the subsystems may be refined, or operationalized, in terms of virtual machines that embody different abstract models of computation. We want to ensure that overall system requirements are achieved in this heterogeneous

implementation. The theory of institutions, general logics, and category theory generally, provide a mathematical framework for ensuring coherence of heterogeneous systems

Section C: *Neural Network Semantics*

To briefly demonstrate the broad applicability of this framework, category theory has even been applied to the semantics of neural networks [Healy]. There is a category of specifications and specification morphisms (called the concept category in this work). A preliminary category has been defined for neural network architectures, called the neural category. There is a functor from the concept category to the neural category that shows how the meaning expressed in a collection of specifications can be mapped into a neural network architecture. Issues such as sensor fusion, or sensor integration, can be dealt with by means of natural transformations. This is illustrated in Figure 9. Details of this preliminary approach can be found in [Healy].

Section D: *Foundation for System Engineering*

In all this work, system requirements can be stated in a category of specifications and specification morphisms, with functors (structure preserving mappings) into virtual machines (which could be as diverse as different programming languages or even neural networks). In general, there is a hierarchy of virtual computers:

1. Actual hardware computer
2. Firmware virtual computer
3. Operating system virtual computer
4. Programming language virtual computer
5. Program virtual computer

Different portions of this hierarchy may be achieved with different models of computation. What we want to be sure to achieve is overall system coherence. To do this, we need to understand how to map the syntax and semantics of these machines. Category theory appears to provide a foundation for doing this.

In a ubiquitous role, category theory appears to provide a foundation for defining, developing, and maintaining complex heterogeneous systems. It is a foundational, mathematical theory for systems development and integration. For corporations such as Boeing that see themselves as integrators of complex systems, it seems worthwhile to develop a research program oriented towards the application of category theory to general systems theory and systems engineering [GoguenSystems].

Chapter 5: Bibliography

1. Astesiano, E. et. al. (eds.), *Algebraic Foundations of Systems Specification*, Springer-Verlag, IFIP State-Of-The-Art Reports, 1999.
2. Bjorner, Dines and Jones, Cliff, *Formal Specification & Software Development*, Prentice-Hall International, 1982.
3. Blaine, Lee and Goldberg, Allen, DTRE – A Semi-Automatic Transformation System, in *Constructing Programs from Specifications*, B. Moller (ed.), North Holland, 1991.
4. Brooking, A., *Corporate Memory: Strategies for Knowledge Management*, Thomson Business Press, 1999.
5. Burstall, R. M. and Goguen, J. A., The Semantics of Clear, a Specification Language, Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification, Lecture Notes in Computer Science, 86, Springer-Verlag, 1980.
6. Coglio, A., et. al., Towards a Provably-Correct Implementation of the JVM Bytecode Verifier, OOPSLA '98 Workshop on the Formal Underpinnings of Java, Vancouver, B.C., October 1998.
7. Crole, Roy, *Categories for Types*, Cambridge University Press, 1993.
8. Czarnecki, K., and Eisenecker, U., *Generative Programming: Methods, Tools, and Applications*, Addison Wesley Publishers, 2000.
9. Denning, P. et. al., *Machines, Languages, and Computation*, Prentice Hall, 1979.
10. Fowler, M. and Scott, K., *UML Distilled – Second Edition*, Addison-Wesley, Object Technology Series, 2000.
11. Gannon, John et al., *Software Specification - A Comparison of Formal Methods*, Ablex Publishing.
12. Goguen, J. A., An Introduction to Algebraic Semiotics, with Applications to User Interface Design, in Computation for Metaphor, Analogy and Agents, edited by Chrystopher Nehaniv, Springer Lecture Notes in Artificial Intelligence, 1999.
13. Goguen, J. A. and Burstall, R. M., Institutions: Abstract Model Theory for Specification and Programming, Journal of the Association of Computing Machinery, 1992.
14. Goguen, J. A., Mathematical Representation of Hierarchically Organized Systems, in *Global Systems Dynamics*, ed. E. Attinger and S. Karger, 1970, pp. 112-128.
15. 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.
16. Healy, M., Category Theory Applied to Neural Modeling and Graphical Representations, in Proceedings of the International Joint Conference on Neural Networks (IJCNN 2000), IEEE Press, New Jersey, July 2000.
17. Jullig, R. and Y. V. Srinivas, Diagrams for Software Synthesis, Proceedings of the 8th Knowledge-Based Software Engineering Conference, Chicago, IL, 1993.
18. Lawvere, W. and Schanuel, S., *Conceptual Mathematics: A First Introduction to Categories*, Cambridge University Press, 1997.
19. MacLane, Saunders, *Categories for the Working Mathematician*, Springer-Verlag, 1971.

20. Mayhew, Debra J., *The Usability Engineering Lifecycle*, Academic Press/Morgan Kauffman, 1999.
21. McConnell, S., *After the Gold Rush: Creating a True Profession of Software Engineering*, Microsoft Press, 1999.
22. Meseguer, Jose, General Logics, Logic Colloquium '87, Eds. Ebbinghaus et al., Elsevier Science Publishers, 1989.
23. Pierce, Benjamin C., *Basic Category Theory for Computer Scientists*, MIT Press, 1994.
24. Rogers, Yvonne, et. al, *Models in the Mind – Theory, Perspective, and Application*, Academic Press, 1992.
25. Sekerinski, E. and Sere, K. (eds.), *Program Development by Refinement: Case Studies Using the B Method*, Springer-Verlag, Formal Approaches to Computing and Information Technology Series, London, 1998.
26. Smith, Doug, Mechanizing the Development of Software, in *Calculational System Design*, Ed. M. Broy NATO ASI series, IOS Press, 1999.
27. Smith, Doug, et. al., Planware – Domain-Specific Synthesis of High-Performance Schedulers, Thirteenth Automated Software Engineering Conference, IEEE Computer Society Press, Los Alamitos, CA, 1998.
28. Smith, Doug, et. al., Synthesis of Schedulers for Planned Shutdown of Power Plants, Eleventh Knowledge-Based Software Engineering Conference, IEEE Computer Society Press, Los Alamitos, CA, 1996.
29. Smith, Doug, KIDS: A Knowledge Based Software Development System, in *Automating Software Design*, Eds. M. Lowry and R. McCartney, MIT Press, 1991.
30. Spivey, J. M., *The Z Notation: A Reference Manual*, Prentice-Hall, New York, 1992.
31. 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.
32. Waldinger, Richard et al., Specware™ Language Manual 2.0.1, Suresoft, Inc, 1996.
33. 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.
34. Warmer, J. and Kleppe, A., *The Object Constraint language – Precise Modeling with UML*, Addison-Wesley, Object Technology Series, 1999.
35. Williamson, K. et. al, Industrial Applications of Software Synthesis via Category Theory - Case Studies Using Specware, Journal of Automated Software Engineering, Kluwer Academic Publishers, 2001.
36. Williamson, K. et. al, Reuse of Knowledge at an Appropriate Level of Abstraction – Case Studies Using Specware, Springer Lecture Notes in Computer Science #1844, International Conference on Software Reuse, 2000.
37. Williamson, K. and Healy, M., Applying Category Theory to Derive Engineering Software from Encoded Knowledge, Springer Lecture Notes in Computer Science #1816, International Conference on Algebraic Methodology and Software Technology, 2000.
38. Williamson, K. and Healy, M., Deriving Engineering Software from Requirements, Journal of Intelligent Manufacturing, Kluwer Academic Publishers, 2000.

39. Williamson, K. and Healy, M., Industrial Applications of Software Synthesis via Category Theory, International Conference on Automated Software Engineering, 1999.
40. Williamson, K. and Healy, M., Formally Specifying Engineering Design Rationale, in Proceedings of the Automated Software Engineering Conference, 1997.
41. Wilson, E., *Consilience: The Unity of Knowledge*, Random House, 1999.
42. Human Engineering Program – Processes and Procedures, US Department of Defense, Handbook MIL-HDBK-46855A, 1996.
43. Widmaier, J, C. Smidts, and X. Huang, "Producing More Reliable Software: Mature Software: Engineering Process vs. State-of-the-Art Technology", University of Maryland Reliability Engineering Program