Designware: Software Development by Refinement

Douglas R. Smith

Kestrel Institute, Palo Alto, California 94304 USA

Abstract

This paper presents a mechanizable framework for software development by refinement. The framework is based on a category of higher-order specifications. The key idea is representing knowledge about programming concepts, such as algorithm design, datatype refinement, and expression simplification, by means of taxonomies of specifications and morphisms.

The framework is partially implemented in the research systems Specware, Designware, and Planware. Specware provides basic support for composing specifications and refinements via colimit, and for generating code via logic morphisms. Specware is intended to be general-purpose and has found use in industrial settings. Designware extends Specware with taxonomies of software design theories and support for constructing refinements from them. Planware builds on Designware to provide highly automated support for requirements acquisition and synthesis of high-performance scheduling algorithms.

1 Overview

A software system can be viewed as a composition of information from a variety of sources, including

- the application domain,
- the requirements on the system's behavior,
- software design knowledge about system architectures, algorithms, data structures, code optimization techniques, and
- the run-time hardware/software/physical environment in which the software will execute.

This paper presents a mechanizable framework for representing these various sources of information, and for composing them in the context of a refinement process. The framework is founded on a category of specifications. Morphisms

are used to structure and parameterize specifications, and to refine them. Colimits are used to compose specifications. Diagrams are used to express the structure of large specifications, the refinement of specifications to code, and the application of design knowledge to a specification.

The framework features a collection of techniques for constructing refinements based on formal representations of programming knowledge. Abstract algorithmic concepts, datatype refinements, program optimization rules, software architectures, abstract user interfaces, and so on, are represented as diagrams of specifications and morphisms. We arrange these diagrams into taxonomies, which allow incremental access to and construction of refinements for particular requirement specifications. For example, a user may specify a scheduling problem and select a theory of global search algorithms from an algorithm library. The global search theory is used to construct a refinement of the scheduling problem specification into a specification containing a global search algorithm for the particular scheduling problem.

The framework is partially implemented in the research systems Specware, Designware, and Planware. Specware provides basic support for composing specifications and refinements, and generating code. Code generation in Specware is supported by inter-logic morphisms that translate between the specification language/logic and the logic of a particular programming language (e.g. CommonLisp or C++). Specware is intended to be general-purpose and has found use in industrial settings. Designware extends Specware with taxonomies of software design theories and support for constructing refinements from them. Planware provides highly automated support for requirements acquisition and synthesis of high-performance scheduling algorithms.

The remainder of this paper covers basic concepts and the key ideas of our approach to software development by refinement, in particular the concept of design by classification [6]. We also discuss the application of these techniques to domain-specific refinement in Planware [1]. A detailed presentation of a derivation in Designware is given in [7]

2 Basic Concepts

2.1 Specifications

A specification is the finite presentation of a theory. The signature of a specification provides the vocabulary for describing objects, operations, and properties in some domain of interest, and the axioms constrain the meaning of the symbols. The theory of the domain is the closure of the axioms under the

spec $Container$ is
  sorts E, $Container$
  op $empty :\rightarrow Container$
  op $singleton : E \rightarrow Container$
  op $\_join\_ : Container, Container \rightarrow Container$
  constructors $\{empty, singleton, join\}$ construct $Container$

  axiom $\forall(x : Container)(x\ join\ empty = x\ \wedge\ empty\ join\ x = x)$
  op $\_in\_ : E, Container \rightarrow Boolean$
  definition of in is
    axiom $x\ in\ empty\ =\ false$
    axiom $x\ in\ singleton(y) = (x = y)$
    axiom $x\ in\ U\ join\ V = (x\ in\ U\ \vee\ x\ in\ V)$
  end-definition
end-spec

Fig. 1. Specification for Containers

rules of inference.

Example: Here is a specification for partial orders, using notation adapted from Specware. It introduces a sort E and an infix binary predicate on E, called le, which is constrained by the usual axioms. Although Specware allows higher-order specifications, first-order formulations are sufficient in this paper.

spec $Partial\text{-}Order$ is
  sort E
  op $\_le\_ : E, E \rightarrow Boolean$
  axiom *reflexivity* is $x\ le\ x$
  axiom *transitivity* is $x\ le\ y\ \wedge\ y\ le\ z\ \Longrightarrow\ x\ le\ z$
  axiom *antisymmetry* is $x\ le\ y\ \wedge\ y\ le\ x\ \Longrightarrow\ x\ =\ z$
end-spec

Example: Containers are constructed by a binary join operator and they represent finite collections of elements of some sort E. The specification shown in Figure 1 includes a definition by means of axioms. Operators are required to be total. The constructor clause asserts that the operators $\{empty, singleton, join\}$ construct the sort *Container*, providing the basis for induction on *Container*.

The generic term *expression* will be used to refer to a term, formula, or sentence.

A model of a specification is a structure of sets and total functions that satisfy the axioms. However, for software development purposes we have a less well-defined notion of semantics in mind: each specification denotes a set of possible implementations in some computational model. Currently we regard these

as functional programs. A denotational semantics maps these into classical models.

2.2 Morphisms

A specification morphism translates the language of one specification into the language of another specification, preserving the property of provability, so that any theorem in the source specification remains a theorem under translation.

A *specification morphism* $m : T \to T'$ is given by a map from the sort and operator symbols of the *domain* spec T to the symbols of the *codomain* spec T'. To be a specification morphism it is also required that every axiom of T translates to a theorem of T'. It then follows that a specification morphism translates theorems of the domain specification to theorems of the codomain.

Example: A specification morphism from *Partial-Order* to *Integer* is:

morphism *Partial-Order-to-Integer* is
  $\{E \mapsto Integer,\ le \mapsto \leq\}$

Translation of an expression by a morphism is by straightforward application of the symbol map, so, for example, the *Partial-Order* axiom $x\ le\ x$ translates to $x \leq x$. The three axioms of a partial order remain provable in *Integer* theory after translation.

Morphisms come in a variety of flavors; here we only use two. An *extension* or *import* is an inclusion between specs.

Example: We can build up the theory of partial orders by importing the theory of preorders. The import morphism is $\{E \mapsto E,\ le \mapsto le\}$.

spec *PreOrder*
  sort E
  op $\_le\_ : E, E \to Boolean$
  axiom *reflexivity* is $x\ le\ x$
  axiom *transitivity* is $x\ le\ y\ \wedge\ y\ le\ z\ \Longrightarrow\ x\ le\ z$
end-spec

spec *Partial-Order*
  import *PreOrder*
  axiom *antisymmetry* is $x\ le\ y\ \wedge\ y\ le\ x\ \Longrightarrow\ x = z$
end-spec

A *definitional extension*, written $A \overset{d}{\rightarrow} B$, is an import morphism in which any new symbol in B also has an axiom that defines it. Definitions have implicit axioms for existence and uniqueness. Semantically, a definitional extension has the property that each model of the domain has a unique expansion to a model of the codomain.

Example: Container can be formulated as a definitional extension of *Pre-Container*:

spec *Pre-Container* is
  sorts E, *Container*
  op $empty : \rightarrow Container$
  op $singleton : E \rightarrow Container$
  op $\_join\_ : Container, Container \rightarrow Container$
  constructors $\{empty, singleton, join\}$ construct *Container*
  axiom $\forall(x : Container)(x\ join\ empty = x\ \wedge\ empty\ join\ x = x)$
end-spec

spec *Container* is
  imports *Pre-Container*
  definition of *in* is
    axiom $x\ in\ empty\ =\ false$
    axiom $x\ in\ singleton(y) = (x = y)$
    axiom $x\ in\ U\ join\ V = (x\ in\ U\ \vee\ x\ in\ V)$
  end-definition
end-spec

A parameterized specification can be treated syntactically as a morphism.

Example: The specification *Container* can be parameterized on a spec *Triv* with a single sort:

spec *Triv* is
  sort E
end-spec

via

parameterized-spec *Parameterized-Container* : $TRIV \rightarrow Container$ is
  $\{E \mapsto E\}$

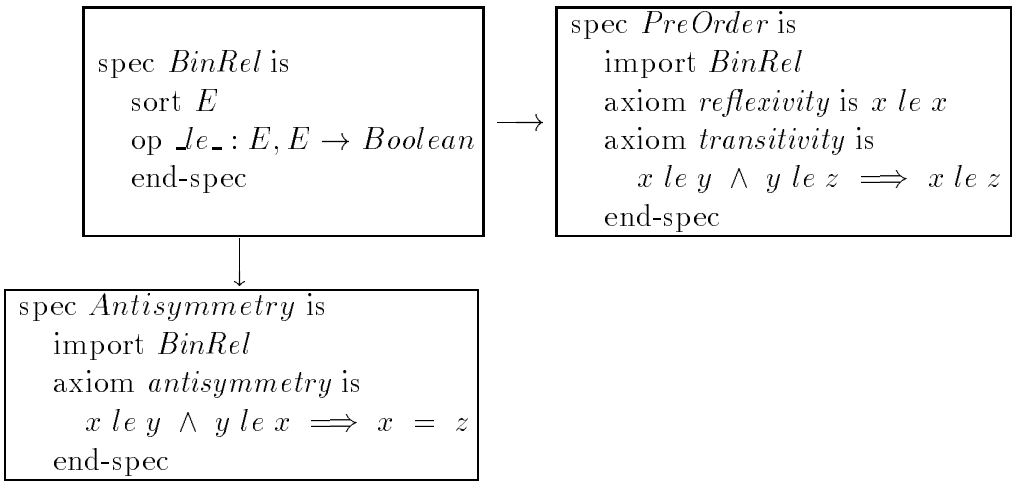
A functorial semantics for first-order parameterized specifications via coherent functors is given by Pavlović [4].

Specification morphisms compose in a straightforward way as the composition of finite maps. It is easily checked that specifications and specification morphisms form a category SPEC. Colimits exist in SPEC and are easily computed. Suppose that we want to compute the colimit of $B \xleftarrow{i} A \xrightarrow{j} C$. First, form the disjoint union of all sort and operator symbols of A, B, and C, then define an equivalence relation on those symbols:

$$s \approx t \text{ iff } (i(s) = t \ \vee \ i(t) = s \ \vee \ j(s) = t \ \vee \ j(t) = s).$$

The signature of the colimit (also known as pushout in this case) is the collection of equivalence classes wrt $\approx$. The cocone morphisms take each symbol into its equivalence class. The axioms of the colimit are obtained by translating and collecting each axiom of A, B, and C.

Example: Suppose that we want to build up the theory of partial orders by composing simpler theories.



The pushout of $Antisymmetry \leftarrow BinRel \rightarrow PreOrder$ is isomorphic to the specification for *Partial-Order* in Section 2.1. In detail: the morphisms are $\{E \mapsto E, \ le \mapsto le\}$ from *BinRel* to both *PreOrder* and *Antisymmetry*. The equivalence classes are then $\{\{E, E, E\}, \ \{le, le, le\}\}$, so the colimit spec has one sort (which we rename E), and one operator (which we rename le). Furthermore, the axioms of *BinRel*, *Antisymmetry*, and *PreOrder* are each translated to become the axioms of the colimit. Thus we have *Partial-Order*.

Example: The pushout operation is also used to instantiate the parameter

in a parameterized specification [2]. The binding of argument to parameter is represented by a morphism. To form a specification for Containers of integers, we compute the pushout of $Container \leftarrow Triv \rightarrow Integer$, where $Container \leftarrow Triv$ is $\{E \mapsto E\}$, and $Triv \rightarrow Integer$ is $\{E \mapsto Integer\}$.

Example: A specification for sequences can be built up from *Container*, also via pushouts. We can regard *Container* as parameterized on a binary operator

spec *BinOp* is
  sort E
  op $\_bop\_ : E, E \rightarrow E$
end-spec

morphism *Container-Parameterization* : $BinOp \rightarrow Container$ is
  $\{E \mapsto E,\ bop \mapsto join\}$

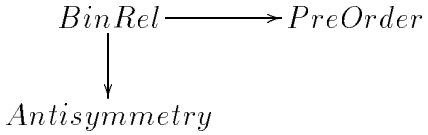
and we can define a refinement arrow that extends a binary operator to a semigroup:

spec *Associativity* is
  import *BinOp*
  axiom *Associativity* is $((x\ join\ y)\ join\ z)\ =\ (x\ join\ (y\ join\ z))$
end-spec

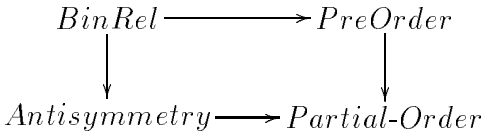
The pushout of $Associativity \leftarrow BinOp \rightarrow Container$, produces a collection specification with an associative join operator, which is *Proto-Seq*, the core of a sequence theory (See Appendix in [7]). By further extending *Proto-Seq* with a commutativity axiom, we obtain *Proto-Bag* theory, the core of a bag (multiset) theory.

2.4 Diagrams

Roughly, a *diagram* is a graph morphism to a category, usually the category of specifications in this paper. For example, the pushout described above started with a diagram comprised of two arrows:



and computing the pushout of that diagram produces another diagram:

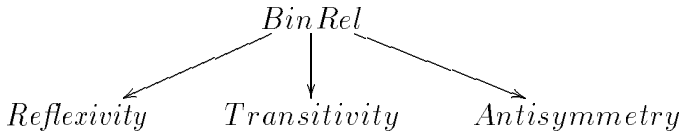


A diagram *commutes* if the composition of arrows along two paths with the same start and finish node yields equal arrows.

2.4.1 The Structuring of Specifications

Colimits can be used to construct a large specification from a diagram of specs and morphisms. The morphisms express various relationships between specifications, including sharing of structure, inclusion of structure, and parametric structure. Several examples will appear later.

Example: The finest-grain way to compose *Partial-Order* is via the colimit of



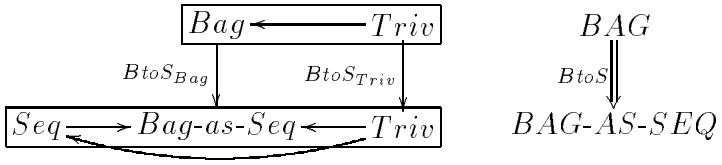
2.4.2 Refinement and Diagrams

As described above, specification morphisms can be used to help *structure* a specification, but they can also be used to *refine* a specification. When a morphism is used as a refinement, the intended effect is to reduce the number of possible implementations when passing from the domain spec to the codomain. In this sense, a refinement can be viewed as embodying a particular design decision or property that corresponds to the subset of possible implementations of the domain spec which are also possible implementations of the codomain.

Often in software refinement we want to preserve and extend the structure of a structured specification (versus flattening it out via colimit). When a specification is structured as a diagram, then the corresponding notion of structured refinement is a diagram morphism. A *diagram morphism* M from diagram D to diagram E consists of a set of specification morphisms, one from each node/spec in D to a node in E such that certain squares commute (a functor underlies each diagram and a natural transformation underlies each diagram morphism). We use the notation $D \implies E$ for diagram morphisms.

Example: A datatype refinement that refines bags to sequences can be pre-

sented as the diagram morphism $BtoS : BAG \implies BAG\text{-}AS\text{-}SEQ$:



where the domain and codomain of $BtoS$ are shown in boxes, and the (one) square commutes. Here *Bag-as-Seq* is a definitional extension of *Seq* that provides an image for *Bag* theory. Specs for *Bag*, *Seq* and *Bag-as-Seq* and details of the refinement can be found in Appendix A of [7]. The interesting content is in spec morphism $BtoS_{Bag}$:

morphism $BtoS_{Bag} : Bag \to Bag\text{-}as\text{-}Seq$ is

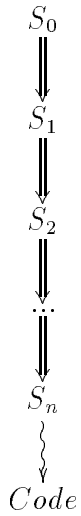
$$\begin{array}{lcl}
\{Bag & \mapsto & Bag\text{-}as\text{-}Seq, \\
empty\text{-}bag & \mapsto & bag\text{-}empty, \\
empty\text{-}bag? & \mapsto & bag\text{-}empty?, \\
nonempty? & \mapsto & bag\text{-}nonempty?, \\
singleton\text{-}bag & \mapsto & bag\text{-}singleton, \\
singleton\text{-}bag? & \mapsto & bag\text{-}singleton?, \\
nonsingleton\text{-}bag? & \mapsto & bag\text{-}nonsingleton?, \\
in & \mapsto & bag\text{-}in, \\
bag\text{-}union & \mapsto & bag\text{-}union, \\
bag\text{-}wfgt & \mapsto & bag\text{-}wfgt, \\
size & \mapsto & bag\text{-}size\}
\end{array}$$

Diagram morphisms compose in a straightforward way based on spec morphism composition. It is easily checked that diagrams and diagram morphisms form a category. Colimits in this category can be computed using left Kan extensions and colimits in SPEC. In the sequel we will generally use the term refinement to mean a diagram morphism.

2.5 Logic Morphisms and Code Generation

Inter-logic morphisms [3] are used to translate specifications from the specification logic to the logic of a programming language. See [8] for more details. They are also useful for translating between the specification logic and the logic supported by various theorem-provers and analysis tools. They are also useful for translating between the theory libraries of various systems.

3 Software Development by Refinement



The development of correct-by-construction code via a formal refinement process is shown to the left. The refinement process starts with a specification S_0 of the requirements on a desired software artifact. Each S_i, $i = 0, 1, ..., n$ represents a structured specification (diagram) and the arrows $\Downarrow$ are refinements (represented as diagram morphisms). The refinement from S_i to S_{i+1} embodies a design decision which cuts down the number of possible implementations. Finally an inter-logic morphism translates a low-level specification S_n to code in a programming language. Semantically the effect is to narrow down the set of possible implementations of S_n to just one, so specification refinement can be viewed as a constructive process for proving the existence of an implementation of specification S_0 (and proving its consistency).

Clearly, two key issues in supporting software development by refinement are: (1) how to construct specifications, and (2) how to construct refinements. Most of the sequel treats mechanizable techniques for constructing refinements.

3.1 Constructing Specifications

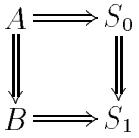
A specification-based development environment supplies tools for creating new specifications and morphisms, for structuring specs into diagrams, and for composing specifications via importation, parameterization, and colimit. In addition, a software development environment needs to support a large library of reusable specifications, typically including specs for (1) common datatypes, such as integer, sequences, finite sets, etc. and (2) common mathematical structures, such as partial orders, monoids, vector spaces, etc. In addition to these generic operations and libraries, the system may support specialized construction tools and libraries of domain-specific theories, such as resource theories, or generic theories about domains such as satellite control or transportation.

3.2 Constructing Refinements

A refinement-based development environment supplies tools for creating new refinements. One of our innovations is showing how a library of abstract refinements can be applied to produce refinements for a given specification. In this paper we focus mainly on refinements that embody design knowledge about (1)

algorithm design, (2) datatype refinement, and (3) expression optimization. We believe that other types of design knowledge can be similarly expressed and exploited, including interface design, software architectures, domain-specific requirements capture, and others. In addition to these generic operations and libraries, the system may support specialized construction tools and libraries of domain-specific refinements.

The key concept of this work is the following: abstract design knowledge about datatype refinement, algorithm design, software architectures, program optimization rules, visualization displays, and so on, can be expressed as refinements (i.e. diagram morphisms). The domain of one such refinement represents the abstract structure that is required in a user's specification in order to apply the embodied design knowledge. The refinement itself embodies a design constraint – the effect is a reduction in the set of possible implementations. The codomain of the refinement contains new structures and definitions that are composed with the user's requirement specification.



The figure to the left shows the application of a library refinement $A \Longrightarrow B$ to a given (structured) specification S_0. First the library refinement is selected. The applicability of the refinement to S_0 is shown by constructing a *classification arrow* from A to S_0 which classifies S_0 as having A-structure by making explicit how S_0 has at least the structure of A. Finally the refinement is applied by computing the pushout in the category of diagrams. The creative work lies in constructing the classification arrow [5,6].

4 Scaling up

The process of refining specification S_0 described above has three basic steps:

(1) select a refinement $A \Longrightarrow B$ from a library,
(2) construct a classification arrow $A \Longrightarrow S_0$, and
(3) compute the pushout S_1 of $B \Longleftarrow A \Longrightarrow S_0$.

The resulting refinement is the cocone arrow $S_0 \Longrightarrow S_1$. This basic refinement process is repeated until the relevant sorts and operators of the spec have sufficiently explicit definitions that they can be easily translated to a programming language, and then compiled.

In this section we address the issue of how this basic process can be further developed in order to scale up as the size and complexity of the library of specs and refinements grows. The first key idea is to organize libraries of specs

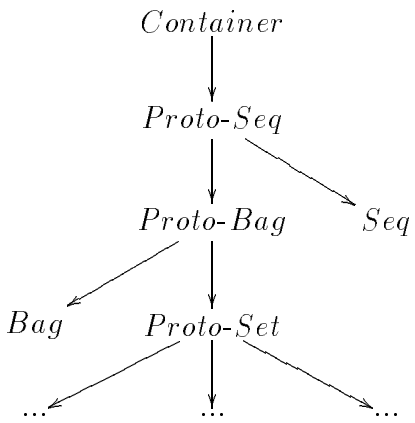


Fig. 2. Taxonomy of Container Datatypes

and refinements into *taxonomies*. The second key idea is to support *tactics* at two levels: theory-specific tactics for constructing classification arrows, and task-specific tactics that compose common sequences of the basic refinement process into a larger refinement step.

4.1 Design by Classification: Taxonomies of Refinements

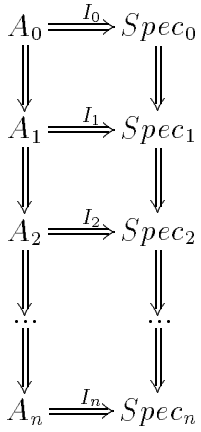
A productive software development environment will have a large library of reusable refinements, letting the user (or a tactic) select refinements and decide where to apply them. The need arises for a way to organize such a library, to support access, and to support efficient construction of classification arrows. A library of refinements can be organized into *taxonomies* where refinements are indexed on the nodes of the taxonomies, and the nodes include the domains of various refinements in the library. The taxonomic links are refinements, indicating how one refinement applies in a stronger setting than another.

Figure 2 sketches a taxonomy of abstract datatypes for collections. Details are given in Appendix A. The arrows between nodes express the refinement relationship; e.g. the morphism from *Proto-Seq* to *Proto-Bag* is an extension with the axiom of commutativity applied to the join constructor of *Proto-Seq*s. Datatype refinements are indexed by the specifications in the taxonomy; e.g. a refinement from (finite) bags to (finite) sequences is indexed at the node specifying (finite) bag theory.

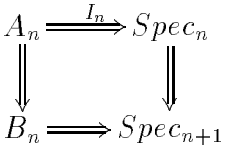
The paper [7] gives a taxonomy of algorithm design theories. The refinements indexed at each node correspond to (families of) program schemes. The algorithm theory associated with a scheme is sufficient to prove the consistency of any instance of the scheme. Nodes that are deeper in a taxonomy correspond to specifications that have more structure than those at shallower levels. Generally, we wish to select refinements that are indexed as deeply

in the taxonomy as possible, since the maximal amount of structure in the requirement specification will be exploited. In the algorithm taxonomy, the deeper the node, the more structure that can be exploited in the problem, and the more problem-solving power that can be brought to bear. Roughly speaking, narrowly scoped but faster algorithms are deeper in the taxonomy, whereas widely applicable general algorithms are at shallower nodes.

Two problems arise in using a library of refinements: (1) selecting an appropriate refinement, and (2) constructing a classification arrow. If we organize a library of refinements into a taxonomy, then the following *ladder construction* process provides incremental access to applicable refinements, and simultaneously, incremental construction of classification arrows.



The process of incrementally constructing a refinement is illustrated in the *ladder construction* diagram to the left. The left side of the ladder is a path in a taxonomy starting at the root. The ladder is constructed a rung at a time from the top down. The initial interpretation from A_0 to $Spec_0$ is often simple to construct. The rungs of the ladder are constructed by a constraint solving process that involves user choices, the propagation of consistency constraints, calculation of colimits, and constructive theorem proving [5,6]. Generally, the rung construction is stronger than a colimit – even though a cocone is being constructed. The intent in contructing $I_i : A_i \Longrightarrow Spec_i$ is that $Spec_i$ has sufficient *defined* symbols to serve as the codomain. In other words, the *implicitly* defined symbols in A_i are translated to *explicitly* defined symbols in $Spec_i$.



Once we have constructed a classification arrow $A_n \Longrightarrow Spec_n$ and selected a refinement $A_n \Longrightarrow B_n$ that is indexed at node A_n in the taxonomy, then constructing a refinement of $Spec_0$ is straightforward: compute the pushout, yielding $Spec_{n+1}$, then compose arrows down the right side of the ladder and the pushout square to obtain $Spec_0 \Longrightarrow Spec_{n+1}$ as the final constructed refinement.

Again, rung construction is *not* simply a matter of computing a colimit. For example, there are at least two distinct arrows from *Divide-and-Conquer* to *Sorting*, corresponding to a mergesort and a quicksort – these are distinct cocones and there is no universal sorting algorithm corresponding to the colimit. However, applying the refinement that we select at a node in the taxonomy *is* a simple matter of computing the pushout. For algorithm design the pushout

simply instantiates some definition schemes and other axiom schemes.

It is unlikely that a general automated method exists for constructing rungs of the ladder, since it is here that creative decisions can be made. For general-purpose design it seems that users must be involved in guiding the rung construction process. However in domain-specific settings and under certain conditions it will possible to automate rung construction (as discussed in the next section). Our goal in Designware is to build an interface providing the user with various general automated operations and libraries of standard components. The user applies various operators with the goal of filling out partial morphisms and specifications until the rung is complete. After each user-directed operation, constraint propagation rules are automatically invoked to perform sound extensions to the partial morphisms and specifications in the rung diagram. Constructive theorem-proving provides the basis for several important techniques for constructing classification arrows [5,6].

4.2 Tactics

The design process described so far uses primitive operations such as (1) selecting a spec or refinement from a library, (2) computing the pushout/colimit of (a diagram of) diagram morphisms, and (3) unskolemizing and translating a formula along a morphism, (4) witness-finding to derive symbol translations during the construction of classification arrows, and so on. These and other operations can be made accessible through a GUI, but inevitably, users will notice certain patterns of such operations arising, and will wish to have macros or parameterized procedures for them, which we call *tactics*. They provide higher level (semiautomatic) operations for the user.

The need for at least two kinds of tactics can be discerned.

(1) *Classification tactics* control operations for constructing classification arrows. The divide-and-conquer theory admits at least two common tactics for constructing a classification arrow. One tactic can be procedurally described as follows: (1) the user selects a operator symbol with a DRO requirement spec, (2) the system analyzes the spec to obtain the translations of the DRO symbols, (3) the user is prompted to supply a standard set of constructors on the input domain D, (4) the tactic performs unskolemization on the composition relation in each Soundness axiom to derive a translations for O_{Ci}, and so on. This tactic was followed in the mergesort derivation.

The other tactic is similar except that the tactic selects constructors for the composition relations on R (versus D) in step (3), and then uses unskolemization to solve for decomposition relations in step (4). This

tactic was followed in the quicksort derivation.

A classification tactic for context-dependent simplification provides another example. Procedurally: (1) user selects an expression *expr* to simplify, (2) type analysis is used to infer translations for the input and output sorts of *expr*, (3) a context analysis routine is called to obtain contextual properties of *expr* (yielding the translation for C), (4) unskolemization and witness-finding are used to derive a translation for *new-expr*.

(2) *Refinement tactics* control the application of a collection of refinements; they may compose a common sequence of refinements into a larger refinement step. Planware has a code-generation tactic for automatically applying spec-to-code interlogic morphisms. Another example is a refinement tactic for context-dependent simplification; procedurally, (1) use the classification tactic to construct the classification arrow, (2) compute the pushout, (3) apply a substitution operation on the spec to replace *expr* with its simplified form and to create an isomorphism. Finite Differencing requires a more complex tactic that applies the tactic for context-dependent simplification repeatedly in order to make incremental the expressions set up by applying the $Expression\text{-}and\text{-}Function \rightarrow Abstracted\text{-}Op$ refinement.

We can also envision the possibility of metatactics that can construct tactics for a given class of tasks. For example, given an algorithm theory, there may be ways to analyze the sorts, ops and axioms to determine various orders in constructing the translations of classification arrows. The two tactics for divide-and-conquer mentioned above are an example.

5 Summary

The main message of this paper is that a formal software refinement process can be supported by automated tools, and in particular that libraries of design knowledge can be brought to bear in constructing refinements for a given requirement specification. One goal of this paper has been to show that diagram morphisms are adequate to capture design knowledge about algorithms, data structures, and expression optimization techniques, as well as the refinement process itself. We showed how to apply a library refinement to a requirement specification by constructing a classification arrow and computing the pushout. We discussed how a library of refinements can be organized into taxonomies and presented techniques for constructing classification arrows incrementally. The examples and most concepts described are working in the Specware, Designware, and Planware systems.

Acknowledgements: The work reported here is the result of extended col-

laboration with my colleagues at Kestrel Institute. I would particularly like to acknowledge the contributions of David Espinosa, LiMei Gilham, Junbo Liu, Duško Pavlović, and Stephen Westfold. This research has been partially supported by the US Air Force Research Lab, Rome NY, and by the Defense Advanced Research Projects Agency.

References

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

[2] Burstall, R. M., and Goguen, J. A. The semantics of clear, a specification languge. In *Proceedings, 1979 Copenhagen Winter School on Abstract Software Specification)*, D. Bjorner, Ed. Springer LNCS 86, 1980.

[3] Meseguer, J. General logics. In *Logic Colloquium 87*, H. Ebbinghaus, Ed. North Holland, Amsterdam, 1989, pp. 275–329.

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

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

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

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

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