Software Development by Refinement

Dusko Pavlovic and Douglas R. Smith

Kestrel Institute, 3260 Hillview Avenue
Palo Alto, California 94304 USA

Abstract. This paper presents an overview of the technical foundations and current directions of Kestrel's approach to mechanizing software development. The approach emphasizes machine-supported refinement of property-oriented specifications to code, based on a category of higher-order specifications. A key idea is representing knowledge about programming concepts, such as algorithm design, and datatype refinement by means of taxonomies of abstract design theories and refinements. Concrete refinements are generated by composing library refinements with a specification.
The framework is partially implemented in the research systems Specware, Designware, Epoxi, 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. Epoxi builds on Designware to support the specification and refinement of systems. Planware transforms behavioral models of tasks and resources into high-performance scheduling algorithms. A few applications of these systems are presented.

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.

B.K. Aichernig and T. Maibaum (Eds.): Formal Methods ..., LNCS 2757, pp. 267–286, 2003.

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, Epoxi 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. Epoxi extends Specware to support the specification and refinement of behavior and the generation of imperative code. Planware transforms behavioral models of tasks and resources into high-performance scheduling algorithms.

The remainder of this paper presents an overview of the technical foundations and current directions of Kestrel's approach to mechanizing software development. A few applications of these techniques are described in Section 6.

2 Basic Concepts

2.1 Specifications

A specification is a 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 rules of inference.

Example: Here is a specification for partial orders, using notation adapted from Specware [18]. 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-Order* is
  sort E
  op $\_le\_ : E, E \to Boolean$
  axiom *reflexivity* is $x\ le\ x$
  axiom *transitivity* is $x\ le\ y\ \wedge\ y\ le\ z\ \Longrightarrow\ x\ le\ z$
  axiom *antisymmetry* is $x\ le\ y\ \wedge\ y\ le\ x\ \Longrightarrow\ x\ =\ y$
end-spec

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.

2.2 Morphisms

A specification morphism translates the language of one specification into the language of another specification while 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. An *interpretation* (between theories) is a slightly generalized morphism that translates symbols to expressions.

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 $x\ le\ y\ \wedge\ y\ le\ z \implies x\ le\ z$
end-spec

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

A *definitional extension*, written $A \xrightarrow{d} B$, 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.

A parameterized specification can be treated syntactically as a morphism. A functorial semantics for first-order parameterized specifications via coherent functors is given by Pavlović [10].

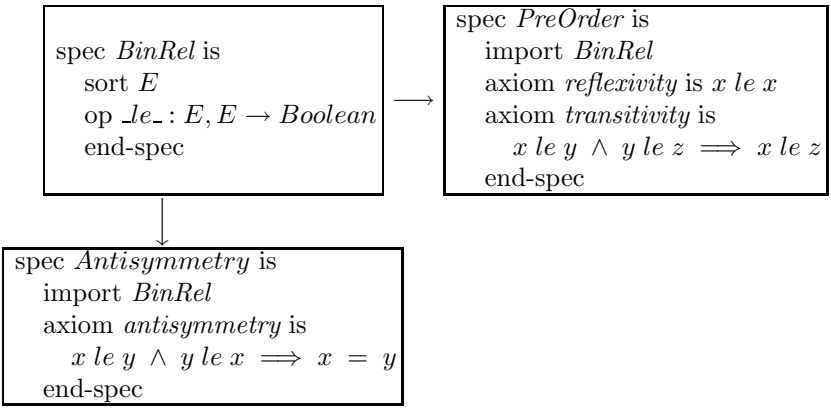
2.3 The Category of Specs

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:

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

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 [3]. 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
  $\{E \mapsto E,\ bop \mapsto join\}$

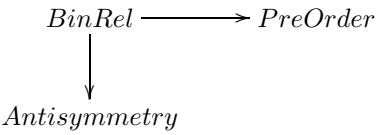
and we can define a refinement arrow that extends a binary operator to a semigroup:

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

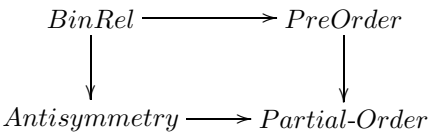
The pushout of $Associativity \leftarrow BinOp \rightarrow Container$, produces a collection specification with an associative join operator, which is *Proto-Seq*, the core of a sequence theory (See Appendix in [16]). By further extending *Proto-Seq* with a commutativity axiom, we obtain *Proto-Bag* 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:



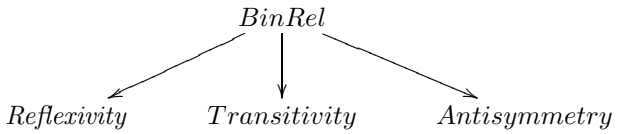
and computing the pushout of that diagram produces another diagram:



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

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.

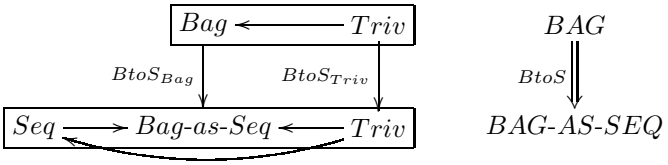
Example: The finest-grain way to compose *Partial-Order* is via the colimit of



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 presented 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 [16]. The interesting content is in spec morphism $BtoS_{Bag}$:

morphism $BtoS_{Bag} : Bag \to Bag\text{-}as\text{-}Seq$ is
$\{Bag \mapsto Bag\text{-}as\text{-}Seq,$
$\ empty\text{-}bag \mapsto bag\text{-}empty,$

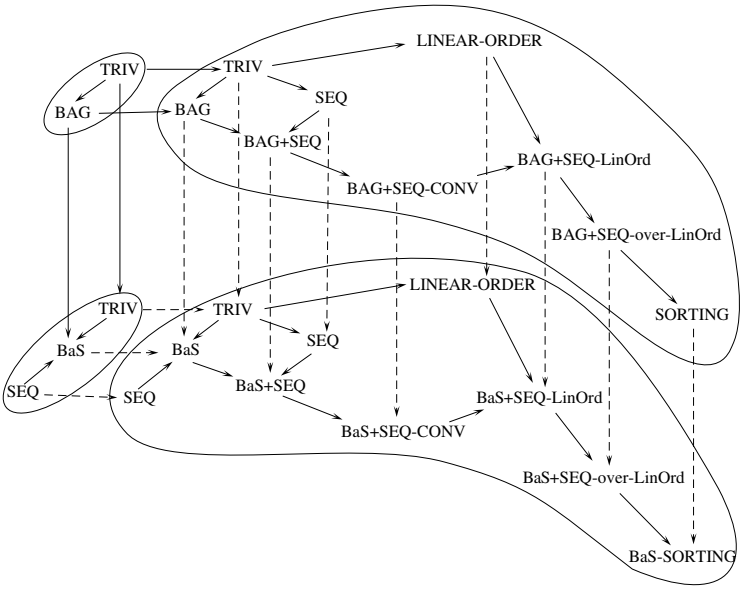


Fig. 1. Refining Bags to Seqs in Sorting.

empty-bag?	$\mapsto$	*bag-empty*?,
nonempty?	$\mapsto$	*bag-nonempty*?,
singleton-bag	$\mapsto$	*bag-singleton*,
singleton-bag?	$\mapsto$	*bag-singleton*?,
nonsingleton-bag?	$\mapsto$	*bag-nonsingleton*?,
in	$\mapsto$	*bag-in*,
bag-union	$\mapsto$	*bag-union*,
bag-wfgt	$\mapsto$	*bag-wfgt*,
size	$\mapsto$	*bag-size*}

Diagram morphisms compose in a straightforward way based on spec morphism composition. It is easily checked that diagrams and diagram morphisms form a category. In the sequel we will generally use the term refinement to mean a diagram morphism.

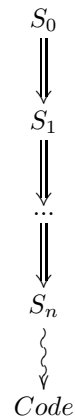
Colimits in this category can be computed using left Kan extensions and colimits in SPEC. Figure 1 shows the pushout of the diagram morphism *BtoS* and a structured specification (diagram) for the problem of sorting a bag over linearly ordered sets (see [16] for details). Intuitively, the universality of the colimit asserts that the resulting diagram is the simplest diagram that refines the sorting diagram and incorporates the refinement information of *BtoS*.

The fact that the colimit calculation constructs a refinement of a given diagram (here the sorting specification) with respect to an abstract refinement (here *BtoS*) is a key tool in our approach to mechanizing the development process.

2.5 Logic Morphisms and Code Generation

Inter-logic morphisms [9] are used to translate specifications from the specification logic to the logic of a programming language. See [18] 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. Section 4 describes mechanizable techniques for constructing refinements.

3.1 Constructing Specifications

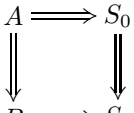
A specification-based development environment supplies tools for creating new specifications and morphisms, for structuring specs into diagrams, and for composing specifications via importation, parameterization, and colimit. In addition, a software development environment needs to support a large library 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 re-

finements 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 [14, 15].

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 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.

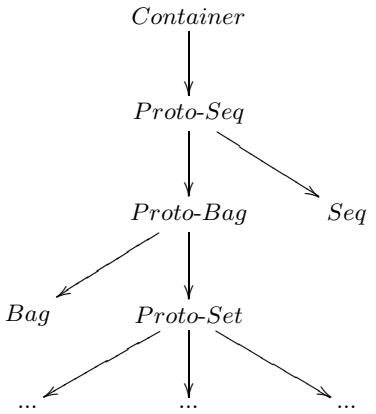


Fig. 2. Taxonomy of Container Datatypes.

Figure 2 sketches a taxonomy of abstract datatypes for collections. 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.

Figure 3 shows 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.

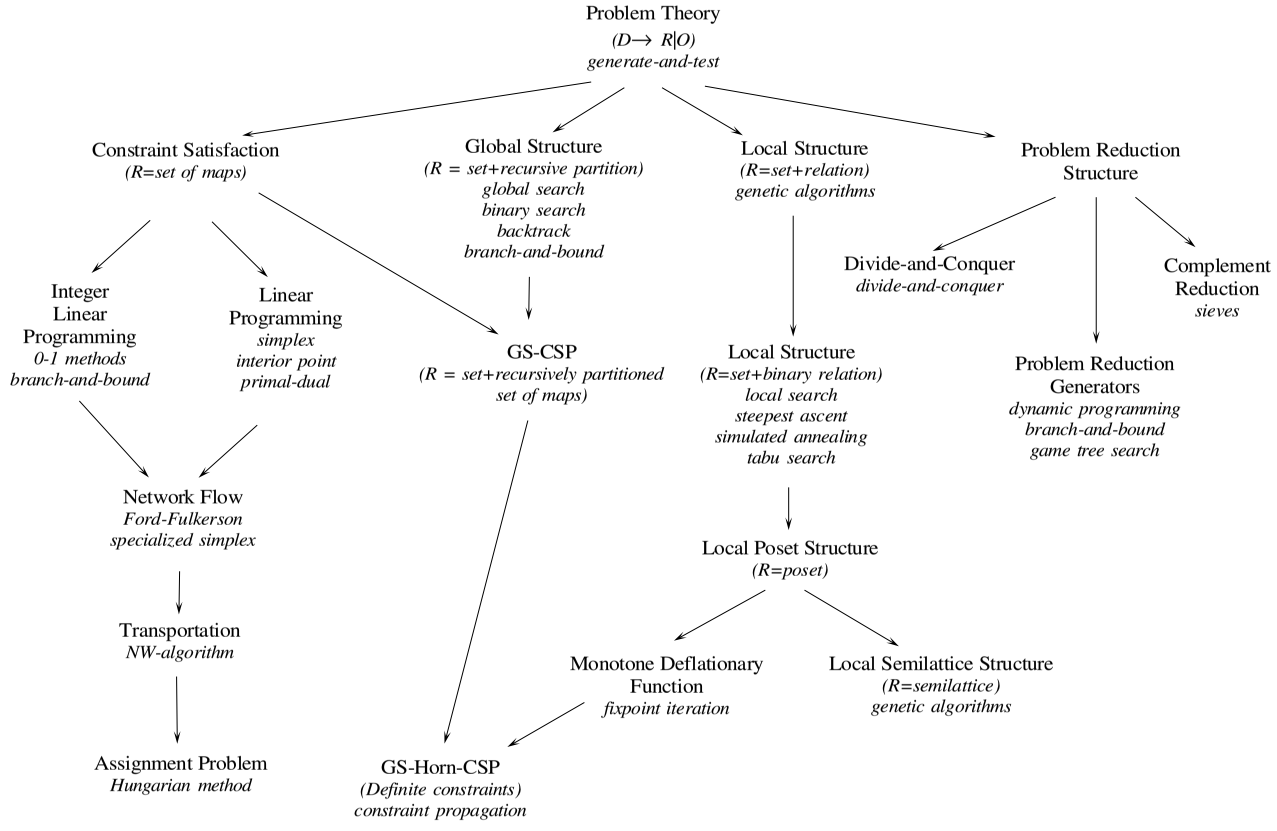
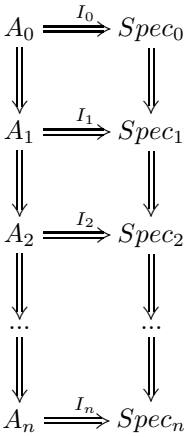
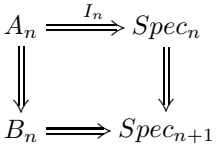


Fig. 3. Taxonomy of Algorithm Theories.

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 [14, 15]. Generally, the rung construction is stronger than a colimit – even though a cocone is being constructed. The intent in constructing $I_i : A_i \Longrightarrow Spec_i$ 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 [14, 15].

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 [2] 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-and-Function* → *Abstracted-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 Specifying Behavior

The results described above most naturally support the development of *functional* programs. To support the specification and development of *concurrent* systems, we felt the need to specify behaviors via some notion of state machine [11].

We are developing an extension of the framework, called *evolving specifications* (or simply *especs*), that supports the specification and development of complex systems. Especs provide the means for explicitly modeling the logical structure and behavior of systems. The framework supports precise, automatable operations for the composition of especs and their refinement. The espec framework is partially implemented in the Epoxi system.

Especs can be seen as a way to naturally extend the Specware/Designware foundation (the category of diagrams of higher-order algebraic specifications) with a combination of the evolving algebras of Gurevich (aka abstract state machines) [6], with the classical axiomatic semantics of Floyd/Hoare/Dijkstra. As in Specware/Designware, especs use morphisms and colimits to support the composition of systems and their refinement to code, but in addition especs provide a natural and novel way to combine logical structure and behavior.

There are four key ideas underlying our representation of state machines as evolving specifications (especs). Together they reveal an intimate connection between behavior and the category of logical specifications. The first three are due to Gurevich [6].

1. *A state is a model* – A state of computation can be viewed as a snapshot of the abstract computer performing the computation. The state has a set of named stores with values that have certain properties.
2. *A state transition is a finite model change* – A transition rewrites the stored values in the state.
3. *An abstract state is a theory* – Not all properties of a state are relevant, and it is common to group states into abstract states that are models of a theory. The theory presents the structure (sorts, variables, operations), plus the axioms that describe common properties (i.e. invariants). We can treat states as static, mathematical models of a global theory thy_A, and then all transitions correspond to model morphisms. Extensions of the global theory thy_A provide local theories for more refined abstract states, introducing local variables and local properties/invariants.

4. *An abstract transition is an interpretation between theories* – Just as we abstractly describe a class of states/models as a theory, we abstractly describe a class of transitions as an interpretation between theories [8, 11]. To see this, consider the correctness of an assignment statement relative to a precondition P and a postcondition Q; i.e. a Hoare triple $P\ \{x\ :=\ e\}\ Q$. If we consider the initial and final states as characterized by theories thy_{pre} and thy_{post} with theorems P and Q respectively, then the triple is valid iff $Q[e/x]$ is a theorem in thy_{pre}. That is, the triple is valid iff the symbol map $\{\ x\ \mapsto\ e\}$ is an interpretation from thy_{post} to thy_{pre}. Note that interpretation goes in the *opposite* direction from the state transition.

The basic idea of especs is to use specifications as state descriptions, and to use interpretations to represent transitions between state descriptions.

The idea that abstract states and abstract transitions correspond to specs and interpretations suggests that state machines are diagrams over Spec^{op}. Furthermore, state machines are composed via colimits, and state machines are refined via diagram morphisms [11]. These concepts are implemented in the Epoxi extension of Specware. Epoxi includes a translator from especs to C code.

Especs support an architectural approach to system design [13]. Components and connectors are represented by parameterized especs where the parameters are the interfaces for components and connectors (ports and roles respectively). The interconnection of components and connectors, forming the architecture, is presented by a diagram in the category of especs (cf.[5, 7]). The colimit of the diagram serves to glue the interfaces together and to form the parallel composition of the constituent behaviors.

In [12], we present a detailed example of a simple system comprised of a radar unit and a mission controller connected by a synchronous communication channel. Especs for the components and the connector specify the structure, behavior, and roles/ports. They are interconnected by means of a diagram and composed via a colimit of especs. A glue-code generator is used to refine the connector, serving to reconcile the data structure mismatches between the two components [4].

6 Applications

We briefly describe several application projects underway at Kestrel that exploit the refinement technology discussed above.

Mission Planning System (MPS). The planning of large-scale cargo transportation missions is one of the most complex scheduling problems in the world. It involves simultaneously scheduling a variety of resources including aircraft, crews, and port facilities as well as supporting resources such as fuel. Other complexities such as routing and diplomatic clearances further complicate any model. Commercial airlines face a related problem, but they are able to separate aircraft and crew scheduling because of the regularities of their service. Kestrel and BBN Technologies have developed a prototype mission planning system for

the US Air Force satisfying most requirements for operation. The main algorithm has been entirely developed and evolved by modifying requirement specifications (stated as pre/post-conditions on the input/output data types), and applying algorithm design, datatype refinement, and optimization tactics. To our knowledge there is no more complex algorithm that has been developed formally from a property-oriented specification and with such as high degree of automation.

Planware. The Planware system is a domain-specific generator of high-performance schedulers [1]. It provides an answer to the question of how to help automate the acquisition of requirements from the user and to assemble a formal requirement specification for the user. The key idea is to focus on a narrow well-defined class of problems and programs and to build a precise, abstract domain-specific specification formalism that covers the class. Interaction with the user is only required in order to obtain the refinement from the abstract spec to a specification of the requirements of the user's particular problem.

To allow users to specify complex multi-resource problems, Planware uses especs to model the behavior of tasks and resources, and uses a service matching theory to handle the interactions of multi-resource problems. For example, a transportation organization might want a scheduler to simultaneously handle its aircraft, crews, fuel, and airport load/unload facilities. Each resource has its own internal required patterns of behavior and may have dependencies on other resources.

The semantics of a resource is the set of possible behaviors that it can exhibit. We treat these behaviors as (temporal) sequences of activities which are modeled as espec modes (abstract states). Each activity has mode variables (e.g. start-time and duration) and any services that it offers (e.g. the flying mode of an aircraft offers transportation service) and services that it requires (e.g. the flying mode of an aircraft requires the services of a crew). A formal theory of a resource should have as models exactly the physically feasible behaviors of the resource. The axioms serve to constrain the values that mode variables can take on in states (e.g. the weight of cargo cannot exceed a maximum bound during the flying mode of an aircraft). The transitions serve to constrain the evolution of the mode variables (e.g. the finish time of one activity must occur no later than the start time of the next activity).

A task is also expressed formally as an espec. The main difference between a task and a resource is that a task offers no service - it only requires services of resources. For example, a cargo container requires transportation service.

We believe that Planware's modeling language is general for expressing scheduling and resource allocation problems. However, the design process currently focuses on the generation of centralized, offline algorithms. The Planware design process has the following steps:

1. Requirement Acquisition – The user supplies a model of a scheduling problem in terms of especs for the kinds of tasks and resources that are of concern. The problem model is formalized into a specification that can be read abstractly as follows: given a collection of task instances (that accord with the task especs) and a collection of resource instances, find a schedule that ac-

complishes as many of the tasks as possible (or (approximately) optimizes the given cost function), subject to all the constraints of the resource models and using only the given resources.
The required and offered services of a resource express the dependencies between resource classes. Planware analyzes the task and resource models to determine a hierarchy of service matches (service required matched with service offered) that is rooted in a task model.

2. Algorithm Design – The problem specification is used to automatically instantiate program schemes that embody abstract algorithmic knowledge about global search and constraint propagation. The algorithm generation process follows the structure of the service hierarchy, resulting in a nested structure of instantiated search schemes.
3. Datatype Refinement and Optimization – Abstract datatypes are refined to concrete programming-language types, and other optimizations are applied.
4. Code generation – Finally code in a programming language (currently CommonLisp) is generated. In one recent example, we developed formal models for air cargo packages, cargo aircraft, air crews, and port facilities (i.e. four espec models). In about one second Planware generates 6560 LOC in our local MetaSlang language, and then translates it to 19088 LOC in Commonlisp comprising over 1780 definitions.

JBV/Applet Generation. Another Kestrel project uses refinement techniques to automate the correct-by-construction generation of secure Java applets from specifications. Previous work developed a formal specification of a Java ByteCode Verifier, and generated correct code from it.

AIM Chip. Motorola Corporation used the Specware tool to produce a successful commercial chip, called the Advanced InfoSec (Information Security) Machine, a VLSI programmable cryptographic processor that was released in 1998. Specware was used to generate and certify the secure kernel of the operating system.

Formal Methods versus CMM Level 4 Process Management. The US Department of Defense sponsored one relatively carefully controlled experimental comparison of two competing development methodologies. One company used the Specware tool and another company used the SEI CMM level 4 process management on the same task and with the same budget. The requirements were expressed in natural language, and the parties were given the same access to domain experts regarding the specification. The formal Specware-based approach was found to result in significantly fewer errors in the final design. Details may be found in [19].

7 Toward Embedded System Design

Current work is extending the modeling capability of especs to the domain of hybrid embedded systems [17]. The ultimate objective is to produce a software

development environment that provides extensive tool support for the development of high-assurance real-time embedded systems from specifications.

The foundation of the development process is a module/interface specification formalism that is abstract, semantically precise, expressive, and provides machine support for key design operations, such as composition (via colimit), refinement (by applying libraries of design knowledge), and reasoning (via general-purpose and specialized inference procedures).

We have attempted to combine in one formalism the ability to precisely specify a module in terms of both the functionality/behavior of its services (based on especs), and their resource constraints (based on the Planware use of especs). Moreover, services are categorized along several dimensions: required versus offered services, and pull services (functions and procedures) versus push services (disseminating state and event information). To our knowledge, there is no other specification formalism that supports compositional reasoning of embedded systems at all levels of abstraction.

Based on this module/interface specification formalism, the framework emphasizes a compositional and refinement-oriented approach to design. The user assembles a high-level model of the physical environment, monitored/controlled system, and the embedded software. Refinements that introduce structure and implementation detail are applied. A key idea is to represent knowledge about embedded system design concepts, and to semiautomatically use those representations to generate refinements. Finally, there is a partitioning process that groups components and connectors that will be mapped to the same target component. The use of specialized resource allocation algorithms can help search for (near)-optimal partitions and maps.

8 Summary

This paper summarizes Kestrel's ongoing efforts to provide practical support for the development of efficient, high-assurance software. The main features of these efforts are:

- *A comprehensive and uniform mathematical foundation* that encompasses requirements (both property-oriented and behavioral), composition, refinement, and code generation, as well as the representation of program/system design knowledge. Building on the category of higher-order specifications, we are developing natural extensions into behavioral specification including features of concurrency, resource constraints, continuity, and stochastic processes.
- *Tool Support* - a practical development formalism must be amenable to extensive automated support, hence our emphasis on a category of syntactic objects (specs) versus semantic objects (structures). The examples and most concepts described are working in the Specware, Designware, Epoxi, and Planware systems. Inference tools such as theorem-provers, constraint propagation, and other analysis tools are critical to providing assurance during refinement.

– *Design Theories* - although it is not emphasized here, a key to practical support for software development from specifications is the reuse of abstract design knowledge. If the ultimate language for communication with computers is a requirements language, then the means must exist to supply the design information that allows the generated system to carry out those requirements (e.g. architectures, algorithms, data structures). Current interest in design patterns, O-O frameworks, and various approaches to generic programming begin to get at this capture of reusable design knowledge, but typically in a way that doesn't relate requirements to code.

Acknowledgments

The work reported here is the result of extended collaboration with our colleagues at Kestrel Institute. We would particularly like to acknowledge the contributions of Matthias Anlauff, Marcel Becker, LiMei Gilham, and Stephen Westfold. This research has been supported by the Office of Naval Research, the US Air Force Research Lab, Rome NY, and by the Defense Advanced Research Projects Agency.

References

1. Becker, M., and Smith, D. R. Planware: Synthesis of resource allocation algorithms. Tech. rep., Kestrel Institute, 2002.
2. 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.
3. 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.
4. Burstein, M., McDermott, D., Smith, D., and Westfold, S. Formal derivation of agent interoperation code. *Journal of Autonomous Agents and Multi-Agent Systems* (2001). (earlier version in Proceedings of the Agents 2000 Conference, Barcelona, Spain, 2000).
5. Goguen, J. A. Categorical foundations for general systems theory. In *Advances in Cybernetics and Systems Research*, F. Pichler and R. Trappl, Eds. Transcripta Books, 1973, pp. 121–130.
6. Gurevich, Y. Evolving algebra 1993: Lipari guide. In *Specification and Validation Methods*, E. Boerger, Ed. Oxford University Press, 1995, pp. 9–36.
7. J.L.Fiadeiro, Lopes, A., and T.Maibaum. Synthesising interconnections. In *Algorithmic Languages and Calculi* (London, 1997), R. Bird and L. Meertens, Eds., Chapman & Hall, pp. 240–264.
8. Kutter, P. W. State transitions modeled as refinements. Tech. Rep. KES.U.96.6, Kestrel Institute, August 1996.
9. Meseguer, J. General logics. In *Logic Colloquium 87*, H. Ebbinghaus, Ed. North Holland, Amsterdam, 1989, pp. 275–329.

10. Pavlovic, D. Semantics of first order parametric specifications. In *Formal Methods '99* (1999), J. Woodcock and J. Wing, Eds., vol. 1708 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 155–172.
11. Pavlovic, D., and Smith, D. R. Composition and refinement of behavioral specifications. In *Proceedings of Sixteenth International Conference on Automated Software Engineering* (2001), IEEE Computer Society Press, pp. 157–165.
12. Pavlovic, D., and Smith, D. R. System construction via evolving specifications. In *Complex and Dynamic Systems Architectures (CDSA 2001)* (2001).
13. Shaw, M., and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, NJ, 1996.
14. Smith, D. R. Constructing specification morphisms. *Journal of Symbolic Computation, Special Issue on Automatic Programming 15*, 5-6 (May-June 1993), 571–606.
15. 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.
16. 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.
17. Smith, D. R. Harbinger: Formal development of embedded systems. Tech. rep., Kestrel Institute, 2002.
18. 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.
19. Widmaier, J., Schmidts, C., and Huang, X. Producing more reliable software: Mature software engineering process vs. state-of-the-art technology? In *Proceedings of the International Conference on Software Engineering 2000* (Limerick, Ireland, 2000), ACM, pp. 87–92.