Composition by Colimit and Formal Software Development [⋆]

Douglas R. Smith

Kestrel Institute, Palo Alto, California 94304 USA

Abstract. Goguen emphasized long ago that colimits are how to compose systems [7]. This paper corroborates and elaborates Goguen's vision by presenting a variety of situations in which colimits can be mechanically applied to support software development by refinement. We illustrate the use of colimits to support automated datatype refinement, algorithm design, aspect weaving, and security policy enforcement.

1 Introduction

Goguen emphasized long ago that colimits are how one composes systems [7]. In particular, Burstall and Goguen focused on specifications as presentations of theories and the composition of specifications by colimit in the CLEAR and CAT system proposals [3, 11]. In a sense this paper serves to corroborate and elaborate Goguen's insight through its applicability to software development by refinement of specifications.

Kestrel's Specware system [29, 12] is a descendant of CLEAR and CAT that uses the cocomplete category of specifications over higher-order logic. Specware is used to support the refinement of specifications into correct code in various target programming languages, including CommonLisp, C, and Java. The role of category theory is to organize the larger-scale structure of specifications and the refinement process. The objects of the category are specifications, diagrams represent structured specifications, and morphisms represent inclusions, parameters, and refinements. Specware uses a colimit algorithm to compose specifications and it uses pushouts to instantiate parameterized specifications (as in [8]). Most of the detailed design work in software development is logical in nature and is performed inside specifications (i.e. below the level of the category). No deep results of category theory are used, but the structuring provided by the categorical framework has been conceptually useful and has guided the implementation of Specware. The Specware system has been used for a variety of applications involving both high assurance (e.g. [4]) properties and high performance (e.g. [1]).

[⋆] in *Algebra, Meaning, and Computation: A Festschrift in Honor of Prof. Joseph Goguen*, Eds. K. Futatsugi, J.P. Jouannaud, and J. Meseguer, Springer-Verlag LNCS 4060, pp. 317-332, 2006.

The most basic and straightforward use of colimits in a category of specifications is to build large specifications out of smaller specifications [2]. We briefly review the technicalities of this usage, but the main focus of the paper is on how to use composition by colimit to construct refinements. In particular, we discuss (1) how to represent design abstractions as specifications and specification morphisms and how to apply a design abstraction by colimit, and (2) how to express some kinds of policy requirements by automata and how to enforce such policies by a suitable colimit. The concepts are illustrated by examples from automated datatype refinement, algorithm design, aspect weaving, and security policy enforcement.

2 Preliminaries

We briefly review the category of specifications over classical higher-order logics, since all the examples and discussion build on it.

A *specification* (or *spec*) 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. For example, the following specification for partial orders is expressed in the MetaSlang specification language of Specware. It introduces a type symbol E and an infix binary predicate on E, called le, which is constrained by the usual axioms.

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

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. In Specware, a specification morphism $m : T \rightarrow T'$ is given by a map from the type and operator symbols of the *domain* spec T to the symbols of the *codomain* spec T'. To be a specification morphism it is sufficient to show 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.

For example, a specification morphism from *Partial-Order* to *Integer* can be presented by:

```
morphism $Partial\text{-}Order\text{-}to\text{-}Integer :\ Partial\text{-}Order \rightarrow Integer$ is
  $\{E \mapsto Integer,\ le \mapsto \leq\}$
```

where *Integer* is a specification for the integers that includes the usual constants (such as 0), comparison relations (such as lesser-or-equal $\leq$), functions (such as addition), and so on.

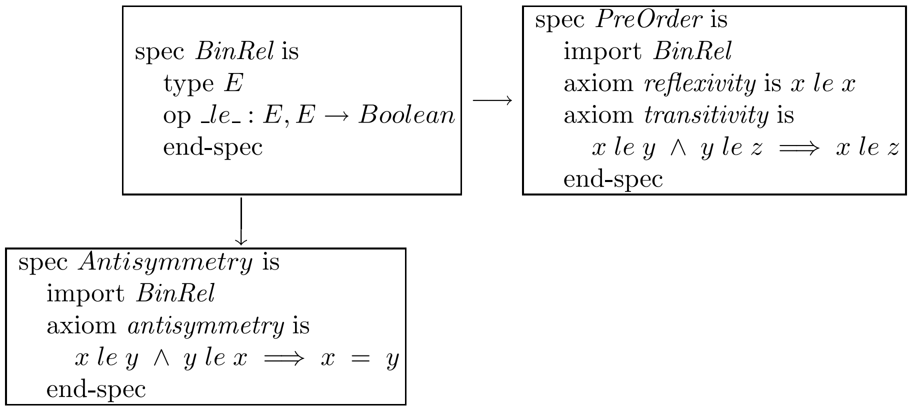
Translation of an expression by a morphism is by straightforward application of the symbol map, so, for example, the *Partial-Order* axiom $\forall(x : E)\ x\ le\ x$ translates to $\forall(x : Integer)\ x \ \leq\ x$ With a reasonable axiomatization of the integers it is easy to verify that the three axioms of a partial order remain provable in *Integer* theory after translation.

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. The colimit can be scalably computed in near-linear time.

For example, suppose that we want to build up the theory of partial orders by composing simpler theories.



spec *BinRel* is
  type E
  op $\_le\_ : E, E \rightarrow Boolean$
  end-spec

spec *PreOrder* is
  import *BinRel*
  axiom *reflexivity* is $x\ le\ x$
  axiom *transitivity* is
    $x\ le\ y\ \wedge\ y\ le\ z\ \Longrightarrow\ x\ le\ z$
  end-spec

spec *Antisymmetry* is
  import *BinRel*
  axiom *antisymmetry* is
    $x\ le\ y\ \wedge\ y\ le\ x\ \Longrightarrow\ x\ =\ y$
  end-spec

The pushout of *Antisymmetry* $\leftarrow$ *BinRel* $\rightarrow$ *PreOrder* is isomorphic to the specification for *Partial-Order* given above. 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 type (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*.

The universal property of the colimit means that there exists a unique specification morphism from the constructed *Partial-Order* specification to any other specification that refines both *PreOrder* and *Antisymmetry.* Intuitively, *Partial-Order* is the simplest specification that composes the logical content of *PreOrder* and *Antisymmetry.*

Although the definitions above are given in higher-order logic, the concepts presented below essentially assume a cocomplete category of specifications over an institution [9].

For purposes of refinement, a loose semantics is natural. Semantics of a refinement morphism is given by a contravariant functor into CAT, the category of small categories. That is, each spec denotes a category of models, and each morphism denotes a functorial mapping that takes each codomain model into a domain model. Particular semantics are enforced by applying appropriate refinements and when performing the institution morphism from the spec language to a programming language.

3 Composing and Refining Specifications

Kestrel's work emphasizes automated tools for the refinement of specifications. There are several reasons for taking this approach to software development: (1) enhanced productivity through automated code generation, (2) enhanced assurance due to the correct-by-construction characteristic of refinement-based derivations, and (3) enhanced software quality and performance due to to automated application of codified best-practice design knowledge.

The first step in developing a new software application in Specware is building a domain specification and capturing the requirements of the application. Composition by colimit plays a major role in building domain specifications. An example from scheduling is shown in Figure 1. Generally, scheduling is about the allocation of resources to tasks so as to satisfy constraints on timeliness, capacity, cost, and so on. In the figure, specifications for *Time* and *Quantity* are shared between *Task* (modeling scheduling tasks) and *Resource* (modeling resources to carry out tasks). *Quantity* is used to model demand in *Task* and to model capacity in *Resource.* A pushout is also used to instantiate a spec SET of finite sets that is parameterized on a base type (called *1-Sort* here). The actual requirements are expressed by input/output constraints (pre/post-conditions) on the scheduler (for more details, see [28]).

In a refinement setting, a formal specification of system requirements is refined to code by incrementally adding design detail. Increments of implementation detail are expressed as morphisms between specifications (in an appropriate category). There is an active community of researchers and practitioners exploring the issues of building requirement specifications out of the (sometimes conflicting) agendas of various stakeholders. What has been missing in this picture is a focus on how to construct refinements – are they mostly ad-hoc, or can they be derived from reusable design abstractions? Most approaches to refinement in the literature (e.g. VDM, Z, RAISE, B) rely on manual invention of refinements,

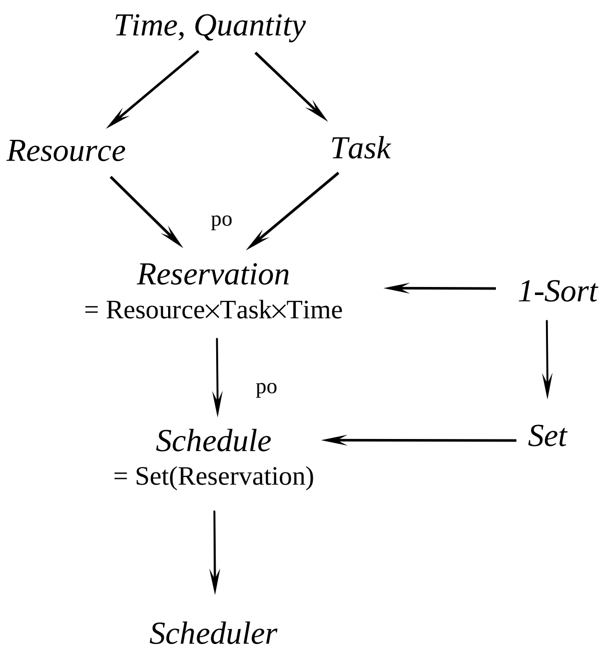


Fig. 1. Scheduling Domain Specification

followed, if desired, by verification of the refinement conditions. Our approach, implemented in KIDS and Specware/Designware, has hypothesized that most code is derived from reusable design abstractions and that these can be codified and mechanically applied. A key component of our research has been collecting and formalizing principles of excellent design practice, as found in algorithm design textbooks and practice, system design patterns/architectures/frameworks, and so on.

The purpose of this paper is to highlight the ways in which colimits, in suitable categories, play a central role in composing these sources of information with the evolving design in order to mechanically generate refinements. Since the colimit is scalably computable in the categories of interest, it can play a central role in a refinement-oriented mechanized system development environment.

4 Design by Classification

Design knowledge typically has two essential components: its content and a characterization of situations in which the content applies. We represent these two components as the codomain and domain of a morphism, respectively. That is, 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. morphisms). The codomain embodies a design constraint – the effect is a reduction in the set of possible implementations. 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 codomain of the refinement contains new structures and definitions that are composed with the user's requirement specification.

$$\begin{tikzcd} A \arrow[r] \arrow[d] & Spec_0 \arrow[d] \\ B \arrow[r] & Spec_1 \end{tikzcd}$$

The figure to the left shows the application of a library refinement $A \to B$ to a given specification $Spec_0$. First the library refinement is selected. The applicability of the refinement to $Spec_0$ is shown by constructing a *classification arrow* from A to $Spec_0$ which classifies $Spec_0$ as having A-structure by making explicit how $Spec_0$ has at least the structure of A. Finally the refinement is applied by computing the pushout. The colimit algorithm generates both the refined specification (the apex shown in the lower right) and the cocone morphisms, including the refinement morphism $Spec_0 \to Spec_1$. The creative work lies in constructing the classification arrow [21, 22].
Furthermore we can organize the design theories into libraries with a taxonomic structure – more general theories refine to more specialized theories. Mechanisms for incrementally accessing and applying design theories from such a library as discussed in [22].

The next two subsections elaborate these notions in the context of datatype refinement and algorithm design respectively.

4.1 Datatype Refinement

Abstract data types (ADTs) allow us to think about data structures in terms of their essential operations and properties. To work effectively with ADTs we must add back in the implementation detail that is abstracted away in a, say, algebraic presentation of an ADT. Refinements serve this purpose. Specifically a morphism between an ADT theory and a (more concrete) datatype theory presents a way to implement the ADT (or dually, a way to view the implementing datatype as the ADT).

Some specific examples includes finite set theory mapping to lists or B-trees or splay trees. Another example: finite sets over a small finite type mapping to hash tables or bit vectors.

Each of these refinements/interpretations can be represented and stored in a library. To apply a datatype refinement we compute the following pushout:

$$\begin{tikzcd} AbstractDT \arrow[r] \arrow[d] & Spec_0 \arrow[d] \\ ConcreteDT \arrow[r] & Spec_1 \end{tikzcd}$$

We sketch a simple example to illustrate the representation of abstract design knowledge as morphisms. In particular, we can refine finite sets to bit vectors as follows. Finite sets over the range 1..32 are partially specified by

```
spec FiniteSet is
    type FSet
    type Elt = 1..32                                % range from 1 to 32
    op {} : FSet                                    % empty set
    op _ with _ : FSet × Elt → FSet                 % add an element
    op _ ∪ _ : FSet × FSet → FSet                   % union
    op _ ∩ _ : FSet × FSet → FSet                   % intersection
    ...
    axiom commutativity is ((S with a) with b) = ((S with b) with a)
    axiom idempotence is ((S with a) with a) = (S with a)
    ...
    end-spec
```

Bit vectors of length 32 are partially specified by

```
spec BitVector32 is
    type BV32
    type Index = 1..32
    op zero : BV32                                  % the zero bit vector
    op set : BV32 × Index → BV32                    % set index bit to 1
    op _ | _ : BV32 × BV32 → BV32                   % bitwise OR
    op _ & _ : BV32 × BV32 → BV32                   % bitwise AND
    op _ << _ : BV32 × Index → BV32                 % left shift
    ...
    end-spec
```

A refinement of *FiniteSet* to *BitVector32* is presented by the morphism

```
morphism FSet-to-BitVector32 is
    {   FSet   ↦   BV32
        Elt    ↦   Index
        {}     ↦   zero
        with   ↦   set
        ∪      ↦   |
        ∩      ↦   &
        ...
    }
```

If we have a specification S that imports *FSet*, then taking a pushout of *FSet-to-BitVector32* with the import morphism $FSet \rightarrow S$ yields a refinement of S in which finite sets are implemented as bit vectors.

As another example, in Specware, the splay tree refinement is the default implementation given to sets due to its good performance profile. Programmers might tempted to avoid working with splay trees since their implementation is a little more complex than simpler representations of sets. A refinement setting allows developers to work with appropriate abstractions *and* obtain good performance.

4.2 Algorithm Design

Just as an algebraic presentation of a datatype aims to capture the abstract essence of the type, an algorithm theory aims to capture the abstract essence of a class of algorithms [27]. For example, consider the class of greedy algorithms (which work to build a solution by iteratively adding the best available component to the incremental solution, until no more components remain). The greedy algorithm can be abstractly represented by a program scheme, which is a definition in a theory that contains partially specified function symbols. A sufficient condition that the scheme generates an optimal solution is given by the matroid property (which is comprised of four conditions; see e.g. [15]). We can represent this package (sufficient structure plus program scheme) as a morphism, prove it once, and store it in a library.

A pushout can be used to apply such an algorithm refinement to a particular problem. For example, the problem of finding a minimum spanning tree can be solved by applying the greedy algorithm theory, yielding Kruskal's algorithm (or Prim's algorithm depending how on the classification arrow is constructed).

$$\begin{tikzcd}
Matroid\ Conditions \arrow[r] \arrow[d] & MST \arrow[d] \\
Greedy\ Scheme \arrow[r] & Kruskal\ Algorithm
\end{tikzcd}$$

This approach to automated algorithm design was first implemented in the KIDS system [20] and more clearly in Specware/Designware [24, 23]. A series of complex high-performance scheduling algorithms for Air Force applications were developed using this approach in KIDS [28] and a domain-specific variant of Designware called Planware [1].

5 Policy Enforcement

The previous two sections describe a means for applying abstract design knowledge to generate algorithms. When we turn to system design, there are issues to contend with that arise less obviously in algorithm design. In particular, cross-cutting concerns are one source of the extra complexity that arises in system design. A concern is cross-cutting if its manifestation cuts across the dominant hierarchical structure of a program. Cross-cutting concerns explain a significant fraction of the code volume and interdependencies of a system. The interdependencies complicate the understanding, development, and evolution of the system.

In this section, we illustrate two forms of cross-cutting concerns and how they can be expressed and mechanically enforced. We call these concerns *policies* to emphasize that (1) they are really requirements, and (2) they tend to reflect non-functional concerns, such as auditing, security, and so on.

The following colimit shows the intention of our approach: to use a colimit in a suitably defined category to enforce policies on a system design.

$$\begin{tikzcd}
\mathit{Shared\ Structure} \arrow[r] \arrow[d] & \mathit{System} \arrow[d] \\
\mathit{Policy} \arrow[r] & \begin{array}{c}\mathit{System\ with} \\ \mathit{Enforced\ Policy}\end{array}
\end{tikzcd}$$

One issue that arises in this context is knowing where the policy applies. For example, a security policy must be applied pervasively in order to provide assurance. In our approach, static analysis [5, 18] is used to find all occurrences and to set up the cospan (i.e. the *Shared Structure* specification above).

5.1 AOP as Invariant Maintenance

A simple example of a cross-cutting concern is an error logging policy – the requirement to log all errors in a system in a standard format. Error logging necessitates the addition of code that is distributed throughout the system code, even though the concept is easy to state in itself.

Aspect-oriented programming (AOP), as exemplified by AspectJ [13], provides a modular way to treat cross-cutting concerns. However, AspectJ aspects are expressed at a programming language level which obscures their intention. The reason for this, of course, is to lower the barriers to usage amongst the broad Java programming community. In [25] we proposed some techniques for specifying cross-cutting concerns as logical invariants to be maintained. For example, to express an error-logging policy as an invariant, we assert that the error-log data structure is equal to the list of all previous errors that have occurred during the course of the computation. To formalize this invariant, we need to reify the history of the computation, purely for specification purposes [25]. The counterpart to aspect weaving is (1) to use static analysis to find all code locations where the invariant might be violated, and (2) to specify and synthesize code to reestablish the invariant. For the error-logging example, static analysis would find all potential code locations where an error might be thrown, and the composition process would compose the throw with an update of the error-log data structure.

By expressing cross-cutting concerns as invariants, we capture their intention more clearly and we can use algorithmic means (static analysis) to determine the complete extent of their application, in contrast to the manual coding of join points in AspectJ.

Our point here is that one of the key mechanisms underlying the enforcement of an invariant is a suitable pushout. To see this most clearly, we switch to a category of abstract state machines over a suitable specification language; e.g. see *especs* in [16, 17]. Here the objects are state machines and the morphisms/refinements represent the simulates relation between automata. An abstract state is given by a specification (for especs we use the higher-order specifications of Specware). For our purposes here, an abstract transition will be specified by a pre/post-condition pair: $\begin{tikzcd} A \arrow[r, dashed, "{[Pre, Post]}"] & B \end{tikzcd}$ (we use dashed arrows for transitions to distinguish them from morphisms in a diagram).

In a category of state machine, particularly especs, refinement means simulation and colimit serves (1) to compose the corresponding state specifications (the pushout of A and C is denoted $A \oplus C$) and (2) to superpose the actions on abstract transitions (the pushout of actions effectively conjoins their effects so the composite action achieves both simultaneously). The following diagram illustrates the composition of one step of the source system $A \dashrightarrow B$ with a step of the policy $C \dashrightarrow D$. The policy asserts that I is to hold invariantly at states and the effect of composition is to add the invariance requirement to the system.

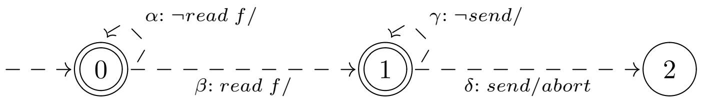
$$\begin{tikzcd}
& \bullet \arrow[dl] \arrow[dr] \arrow[r, dashed] & \bullet \arrow[dl] \arrow[dr] & \\
A \arrow[r, dashed, "{[Pre,Post]}"] \arrow[dr] & B \arrow[dr] & C \arrow[r, dashed, "{[I,I]}"] \arrow[dl] & D \arrow[dl] \\
& A \oplus C \arrow[r, dashed, "{[Pre \wedge I, Post \wedge I]}"'] & B \oplus D &
\end{tikzcd}$$

Static analysis is used to find the association of system steps and policy steps, then a pushout, as above, is used to compose the two. A further synthesis step is needed in order to synthesize an action that achieves the composed action specification $[Pre \wedge I, Post \wedge I]$. For a variety of detailed examples see [25].

5.2 Enforcing Automata-based Security Policies

The previous section described a simple kind of policy, based on invariant state properties, and the composition and synthesis mechanisms that underlie enforcement. A more general kind of policy can be specified by means of automata or by temporal logic formulas.

As a concrete example, consider the following simple security policy which is adapted from Schneider [19]. Whenever a process reads from a particular file f, it is not allowed to send any messages. The policy states a particular kind of information flow constraint. The policy can be expressed as a policy automaton:



The transitions are labeled in the form *name* : *event/action*. The events are expressed as source-code patterns that either succeed (with bindings of pattern variables) or fail. If an action is omitted, then it is a no-op. This policy has only has one prescription of an action to take in a particular context – in policy state 1, if a *send* is attempted, then abort the program. The effect of enforcement will be to terminate any behaviors that do not implement the policy (a send following a read of file *f*). For examples of the enforcement of automata-based policies that prescribe behavior, see the error-handling policies in [26].

Colimits can be used to enforce policies specified by a policy automaton. However, there are interesting issues that arise. The foremost is that the effect of enforcing this policy is to sometimes cause the program to abort (terminate abnormally) when the system would otherwise continue normally. There are two problems here: (1) how to handle conflicting constraints on the system (here the system may satisfy constraints that conflict with the policy), and (2) how to define an appropriate notion of refinement (morphism) that allows termination of behaviors.

One approach to handling conflicting requirements is to treat system requirements as having a linear priority order. The idea is that a system satisfies a priority ordering of constraints if whenever the system fails to satisfy one constraint C, then it must satisfy some other higher-priority constraint. For example, it is often the case in system code that safety and security constraints dominate functional constraints. We make this approach more precise in the following.

Let $\langle R, \prec \rangle$ be a linearly ordered set of temporal formulas [14], and S a program. We say that a behavior b satisfies R if for each formula F in R, either S satisfies F or it satisfies some other formula $G \in R$ such that $F \prec G$. S satisfies R if every behavior of S satisfies R.

Technically, there is no extra expressive power in priority-ordered requirements. Consider the simplest situation in which there are just two requirements A and B together with the order $A \prec B$. An equivalent specification has the two requirements $A \vee B$ and B without an order. Clearly this notion of satisfaction is weak, since it admits programs that satisfy B but not A. While one could pursue this to obtain a stronger theoretical definition of satisfaction (e.g. by considering maximal satisfaction of dominated constraints), we take a pragmatic approach that addresses the problem via the design process. That is, our approach will be to perform design starting with the bottommost requirements of the order – typically these are the basic functional constraints. Then, we iteratively select dominating requirements in order and enforce them by colimit in the evolving design. In this way, whenever we enforce a requirement, the composition process will only override dominated constraints. The result is a design that will tend to satisfy the base functionality requirements as much as possible, but with some behaviors that accord with overriding policy constraints.

Mobile code provides a clear scenario in which this bottom-up design approach makes sense. Mobile code typically cannot be designed to anticipate all environments that it might run in. One host environment may have local policies that must be enforced, and it can do so by, say, composing the policies at the byte-code level at upload time. This way, the local environment's policies are maintained even if it means disallowing behaviors of the mobile code that might be acceptable in other environments.

Our point here is not to fully define a new approach to program satisfaction, nor a new design methodology, but simply to show another context in which composition by colimit provides basic support to system development.

The second problem mentioned above, a suitable notion of refinement that allows behavior termination, can be addressed as follows. In the category of especs [16, 17], abstract states are given by specifications and abstract transitions are modeled by suitable morphisms – a state machine is then a diagram over a category of specs. Each abstract state naturally has the identity self-transition which is the identity morphism on specs. Semantically, the behaviors of such an espec includes arbitrary stuttering (no-op transitions that do not change state). In the literature, behaviors that stutter are often ruled out, although they play a crucial role in refinement. We propose to go farther and admit all such stuttering behaviors, including behaviors in which the machine stutters forever on some state. There are at least two reasons to adopt this rather loose semantics. First, it allows us to model failure in the underlying computation substrate. Most formal models of behavior assume a perfect computational model and ignore the unreliability of the hardware/software platform on which software executes. Second, it allows us to treat as refinement the notion of policy enforcement that works by terminating bad behaviors. In both cases, the idea is that for any behavior that successfully reaches a final/accepting state (or does so infinitely often), the semantics also includes all prefixes of that behavior. Each proper prefix corresponds to a computation that is terminated (due to failure of computational service, or to policing action, etc.). As a consequence, we obtain the conventional notion of trace-containment semantics for refinement. That is, every behavior of the codomain machine (including abnormally terminated behaviors) maps-to/simulates a behavior of the domain machine.

Enforcement of a policy automaton occurs in two stages. In the first stage, static analysis is used to simulate the automaton by matching the event patterns against the control-flow of the system source code. Recent progress has produced scalable low-order polynomial time algorithms for policy simulation [10, 6]. These algorithms work by simulating the policy forward through the source code, recording the policy states and transitions in labels on the control-flow graph of the source code. When matching a policy transition labeled *event*/*action*, if the event pattern matches a source-code transition, then the policy transition (instantiated with the bindings from the match) is associated with the source transition. The algorithms terminate when a fixpoint is reached.

In effect, static analysis creates a refinement of the policy automaton that has the same essential shape as the source code, thus enabling automatic composition by colimit.

Consider for example the code

```
int c;
if c=0 then read f;
send m;
...
```

which is represented by a state machine in Figure 2. The figure also shows the results of policy simulation/analysis – each state of the code is labeled with the states of the policy automaton that it could possibly be in for some input, and each transition is labeled with the set of possible policy transitions that it simulates for some input.

$$\begin{tikzcd}
\bullet \arrow[d, dashed] \\
\bullet \arrow[d, dashed, bend right, "c\neq 0\vdash"'] \arrow[d, dashed, bend left, "c=0 \vdash read\ f"] \\
\bullet \arrow[d, dashed, "send\ m"] \\
\bullet
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
\{0\} \arrow[d, dashed, "\{\alpha\}"] \\
\{0\} \arrow[d, dashed, bend right, "\{\beta\}"'] \arrow[d, dashed, bend left, "\{\gamma\}"] \\
\{0,1\} \arrow[d, dashed, "\{\beta\}"] \\
\{0\}
\end{tikzcd}$$

Fig. 2. Results of Static Analysis

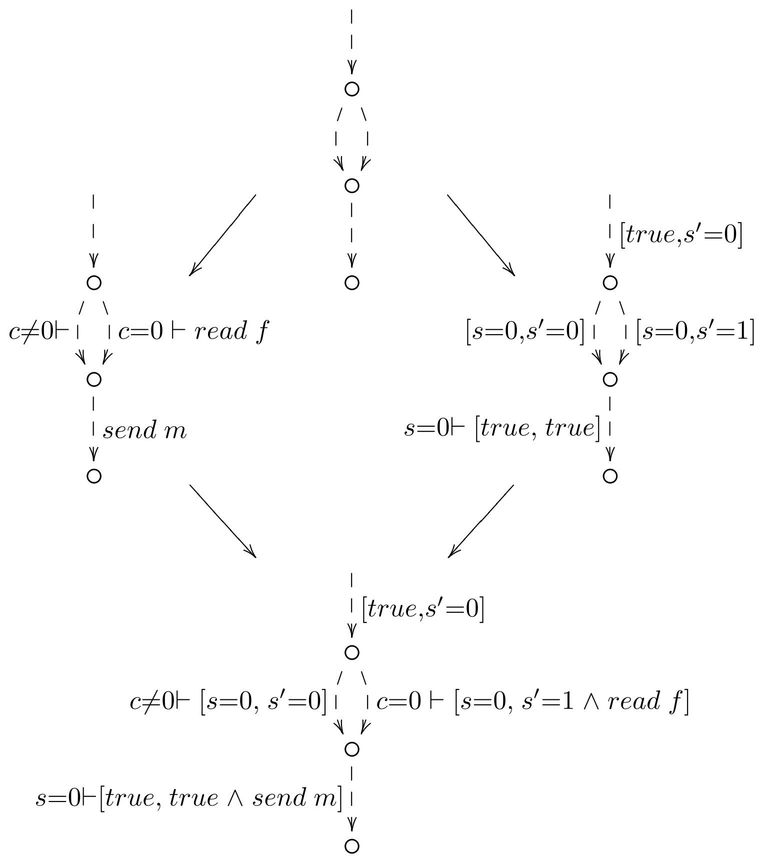


Fig. 3. Colimit to Enforce Policy

Conceptually, the static analysis sets up a cospan in the category of especs [16, 17]. Figure 3 shows both the cospan and the cocone. The static analysis allows us to set up a refinement of the policy automaton (shown on the right of the cospan) and the abstract shape that is common to the source code and the policy instance. In the example, the key feature is the policy ambiguity that results from the conditional: after the conditional, the system state is in either policy state 0 or 1 depending on which branch was taken. Crucially then, the `send` command is either (i) acceptable, if the policy state is 0, or (ii) forbidden if the policy state is 1. The policy instance automaton reflects this by recording the policy transitions that correspond to system transitions.

Computing the pushout has the essential effect of enforcing the security policy in the source code. Finally, program synthesis processes are applied to the pushout specification and the result is translated back to the following source-level code:

```
int c;
int s;  /* state variable */
s := 0;
if c=0
   then {read f || s := 1}
if s=0
   then send m
   else abort;
...
```

In the example, the composition results in the code aborting when $c = 0$. The pushout object is a refinement of both the policy and the source code.

The approach outlined above applies to a given software design, and has the effect of aborting behaviors that are forbidden by policy. while we can formulate this process in terms of pushouts in an appropriate category, there are pros and cons to this approach. It makes sense to use this approach with code of unknown provenance that must be made to conform to local policies (e.g. mobile code or services supplied over the Internet). However for bespoke code, the framework gives the developer too much freedom – it doesn't provide incentives for the programmer to find ways to satisfy both the functional requirements as well as safety and security policies. Our view is that good designers will develop an architecture that supports for the kinds of policies that can be expected for the system. The effect then of policy enforcement would be to add in the details of the policy to the appropriate architectural mechanisms. A good example is access control. There are standard architectures for access control [30] that prescribe the mediation of a guard in any access to a resource that requires some protection. The design pattern puts the requisite structure in place and the colimit composes in the policy details.

6 Concluding Remarks

Our goal has been to show how composition by colimit can play a fundamental role in software development by refinement. The benefits of these foundations include enhanced productivity through automated code generation, enhanced assurance due to the correct-by-construction characteristic of refinement-based derivations, and potentially enhanced software quality and performance due to to automated application of codified best-practice design knowledge.

Acknowledgments: This work was partially supported by the US Department of Defense and by the Office of Naval Research under Grant N00014-04-1-0727.

References

1. Becker, M., Gilham, L., and Smith, D. R. Planware II: Synthesis of schedulers for complex resource systems. Tech. rep., Kestrel Technology, 2003.
2. Burstall, R. M., and Goguen, J. A. Putting theories together to make specifications. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* (Cambridge, MA, August 22–25, 1977), IJCAI, pp. 1045–1058.
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. Coglio, A. Toward automatic generation of provably correct Java Card applets. In *Proc. 5th ECOOP Workshop on Formal Techniques for Java-like Programs* (July 2003).
5. Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1977), ACM, pp. 238–252.
6. Das, M., Lerner, S., and Seigle, M. ESP: Path-sensitive program verification in polynomial time. In *SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)* (2002).
7. 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.
8. Goguen, J. A. Parameterized programming. *IEEE Transactions on Software Engineering SE-10*, 5 (September 1984), 528–543.
9. Goguen, J. A., and Burstall, R. M. Institutions: Abstract model theory for computer science. *Journal of the ACM 39*, 1 (1992), 95–146.
10. Hallem, S., Chelf, B., Xie, Y., and Engler, D. A system and language for building system-specific, static analyses. In *SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'02)* (2002).
11. J. Goguen and R. Burstall. CAT: a system for the structured elaboration of correct programs from structured specifications. Tech. Rep. CSL-118, SRI International, 1988.
12. Kestrel Institute. *Specware System and documentation*, 2003. http://www.specware.org/.
13. Kiczales, G., and et al. An Overview of AspectJ. In *Proc. ECOOP, LNCS 2072, Springer-Verlag* (2001), pp. 327–353.

14. Manna, Z., and Pnueli, A. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, 1992.
15. Papadimitriou, C. H., and Steiglitz, K. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, Englewood Cliffs, NJ, 1982.
16. Pavlovic, D., and Smith, D. R. Composition and refinement of behavioral specifications. In *Proceedings of Automated Software Engineering Conference* (2001), IEEE Computer Society Press, pp. 157–165.
17. Pavlovic, D., and Smith, D. R. Evolving specifications. Tech. rep., Kestrel Institute, 2004.
18. Reps, T., Horwitz, S., and Sagiv, M. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the Twenty-Second ACM Symposium on Principles of Programming Languages* (1995), ACM, pp. 49–61.
19. Schneider, F. Enforceable security policies. *ACM Transactions on Information and System Security 3*, 1 (February 2000), 30–50.
20. Smith, D. R. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16*, 9 (1990), 1024–1043.
21. Smith, D. R. Constructing specification morphisms. *Journal of Symbolic Computation, Special Issue on Automatic Programming 15*, 5-6 (May-June 1993), 571–606.
22. Smith, D. R. Toward a classification approach to design. In *Proceedings of Algebraic Methodology and Software Technology (AMAST)* (1996), vol. LNCS 1101, Springer-Verlag, pp. 62–84.
23. Smith, D. R. Designware: Software development by refinement. In *Proceedings of the Eighth International Conference on Category Theory and Computer Science* (1999), M. Hoffman, D. Pavlovic, and P. Rosolini, Eds., pp. 355–370.
24. 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.
25. Smith, D. R. A generative approach to aspect-oriented programming. In *Proceedings of the Third International Conference on Generative Programming and Component Engineering* (2004), Springer-Verlag LNCS 3286, pp. 39–54.
26. Smith, D. R., and Havelund, K. Automatic enforcement of error-handling policies. Tech. rep., Kestrel Technology, September 2004.
27. Smith, D. R., and Lowry, M. R. Algorithm theories and design tactics. *Science of Computer Programming 14*, 2-3 (October 1990), 305–321.
28. Smith, D. R., Parra, E. A., and Westfold, S. J. Synthesis of planning and scheduling software. In *Advanced Planning Technology* (1996), A. Tate, Ed., AAAI Press, Menlo Park, pp. 226–234.
29. 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.
30. The Open Group. Security design patterns. Tech. rep., http://www.opengroup.org/security/gsp.htm, 2004.