Composition and Refinement of Behavioral Specifications

Dusko Pavlovic and Douglas R. Smith
Kestrel Institute
3260 Hillview Avenue
Palo Alto, California 94304 USA

Abstract

This paper presents a mechanizable framework for specifying, developing, and reasoning about complex systems. The framework combines features from algebraic specifications, abstract state machines, and refinement calculus, all couched in a categorical setting. In particular, we show how to extend algebraic specifications to evolving specifications (especs) in such a way that composition and refinement operations extend to capture the dynamics of evolving, adaptive, and self-adaptive software development, while remaining efficiently computable. The framework is partially implemented in the Epoxi system.

1 Introduction

How can we make the construction of complex systems easier and more reliable? To get a handle on the complexity, many researchers and engineers advocate an architectural approach to system design: a system is treated as a composition of components together with the connectors that mediate their interaction (e.g. see [10]). Sometimes however, the goal of having a clear, simple architecture is at odds with performance goals for the system. A way out of this dilemma is to work toward a framework that allows the *composition* of components and connectors in a high-level architectural design, followed by the *refinement* of the design to code. The refinement process may break down component and connector boundaries to optimize system performance, as well as introducing lower-level design decisions (such as subsystem architectures, algorithms, and data structures).

This paper introduces a formal 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 grew out of higher-order algebraic specifications as implemented in Specware [11], the evolving algebras of Gurevich (aka abstract state machines) [4], as well as the classical axiomatic semantics of Floyd/Hoare/Dijkstra. Especs go beyond all three, not only allowing the capture of logical structure and behavior, but also the composition of systems and their refinement to code. Of course the composition and refinement operations are meaning-preserving, so that any code produced by means of composition and refinement is guaranteed to be consistent with the initial especs.

The paper is structured straightforwardly. We first discuss how to extend logical specifications to model behavior, and then define especs and how to refine and compose them formally. These concepts are illustrated by simple examples. This paper presumes some knowledge of basic category theory (see [2, 11] for relevant background). More details about especs may be found in [8]. Related approaches to providing categorical foundations for specifying, composing and refining behaviors may be found in [3, 5].

2 From Logical Theories to State Machines/Behaviors

EPOXI is made of two basic building blocks: theories and translations (also known as theory morphisms).

- A **theory** formalizes, in predicate logic, what is known about a domain, or an artifact in general. A theory is comprised of a language and a subset of the language called theorems.
- A **translation** is a morphism between theories: it maps the language (signature) of one theory into the terms of another one, while preserving their meaning and validity: type structure is preserved, and the theorems are mapped to theorems. A translation is presented by a map from

the symbols in one theory to expressions in another. The map is applied recursively to translate expressions.

On this foundation we can formally model state machines. A state of computation can be viewed as a snapshot of the abstract computer performing the computation. A rudimentary computer can be viewed as a set of stores, with an abstract mechanism rewriting the stored values. The rewrites are the computation steps, or transitions. As in many logical formalisms for behavior (cf. [4, 6], we treat states as (static, mathematical) *models* of a global theory thy_A, and transitions as finite changes to the components of a state/model. For example, an array is represented as a finite function, whose value may vary over the possible models. A transition could correspond to an assignment that changed the array/finite-function. The computation of a program specified by thy_A, evolving from state to state, can be envisioned as "jumping" from model to model, in $\mathsf{Mod}(\mathsf{thy}_A)$.

To see the connection between theories and translations on one hand, and states and transitions on the other, 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 $\{\ x\ \mapsto\ e\}$ induces a translation from thy_{post} to thy_{pre}. Note that the translation goes in the *opposite* direction from the state transition.

In practice however, one usually deals with abstract states rather than individual states/models. In reasoning about programs, we are typically interested in states that satisfy certain properties, so we use specifications as general *state descriptions*, and pass from models to sets of models that are specified by extensions of the global spec.

The basic idea of especs is to use specifications (finite presentations of a theory) as state descriptions, and to use translations to represent abstract transitions between state descriptions.

The specification of each state description corresponds to its local structure and properties/invariants. The specification common to all state descriptions specifies the global structure and invariants of the system. Any structure that is common to all states that a computation can reach is formalized as a (global) specification; the common structure includes variables and their sorts, as well as axioms (global invariants) and operations (global constants).

Example

Let us see a simple program in this framework. Here, `stad` denotes a state description, `step` a transition. The espec GCD-0 defines the concept of the greatest-common-divisor of two natural numbers and the state machine specifies the required behavior of a greatest-common-divisor computation.

```
espec GCD-base is
  spec                      ;; the keyword spec encloses the logical specification
    const X-in,Y-in : Pos   ;; X-in and Y-in are constant positive integers
    var Z : Pos             ;; Z is a positive integer that varies over states

    op gcd : Pos, Pos -> Pos
    axiom gcd-spec is           ;; this axiom specifies the gcd problem
        gcd(x,y) = z => (divides(z,x) & divides(z,y)
                         & forall(w:Pos)(divides(w,x) & divides(w,y) => w <= z))
  end-spec

  prog       ;; the keyword prog encloses the state machine (empty in this case)
  end-prog

end-espec
```

```
espec GCD-0 is
  import GCD-base
  spec                    ;; the spec extends the spec from GCD-base with a theorem
    thm gcd(x,x) = x      ;; this theorem follows from axiom gcd-spec
  end-spec

  prog                    ;; the keyword prog encloses the state machine
    stad One init[X-in,Y-in] is    ;; the initial state receives X-in and Y-in
    end-stad

    stad Two fin[Z] is     ;; this stad extends the global spec with a local axiom
       axiom Z = gcd(X-in,Y-in)
    end-stad

    step Out : One -> Two is   ;; transition from stad One to stad Two
      Z |-> gcd(X-in,Y-in)
    end-step
  end-prog
end-espec
```

Note that the steps are expressed in terms of symbol translations. Because of the connection between translations and transitions, we will henceforth use assignments instead; i.e. write `x := e` instead of `x |-> e`.

Espec `GCD-1`, below, refines `GCD-0`. The prog expresses the classical GCD algorithm, which might have been generated by a design tactic. `GCD-1` extends the logical spec of `GCD-0` with two local variables `X` and `Y`. Essentially, the refinement adds a new stad and two looping transitions that preserve the key loop invariant of the program: `X` and `Y` change under the transitions, but always so that their GCD is the same as the GCD of the input values `X-in` and `Y-in`.

```
espec GCD-1 is
  import GCD-base
  spec ;; two new vars used to compute GCD
    var X,Y : Pos
  end-spec

  prog
    stad One init[X-in,Y-in] is
    end-stad

    stad Loop is
      axiom gcd(X-in,Y-in) = gcd(X,Y)
    end-stad

    stad Two fin[Z] is
      axiom Z = X
      axiom X = Y
      axiom Z = gcd(X-in,Y-in)
    end-stad

    step initialize : One -> Loop is
      X := X-in
      Y := Y-in
    end-step

    step Loop1 : Loop -> Loop is
      X>Y -> X := X - Y
    end-step

    step Loop2 : Loop -> Loop is
      Y>X -> Y := Y - X
    end-step

    step Out : Loop -> Two is
      X=Y -> Z := X
    end-step
  end-prog
end-espec
```

It is straightforward to check that `GCD-1` is internally consistent; e.g. to show that `Loop1` corresponds to a translation, we must show

$$Loop,\ X > Y \ \vdash\ gcd(X\text{–}in, Y\text{–}in) \ = \ gcd(X - Y, Y)$$

The correctness conditions of refinements is addressed in Section 4.

3 Especs

The concept of espec is now formally defined.

Definition 3.1 *A* graph s *consists of two sets* edge_s *and* node_s*, and two functions,* dom_s *and* cod_s *from* edge_s *to* node_s*.*

A shape *is a graph* s*, which is moreover*

- *reflexive, in the sense that there is a function* $\mathrm{id}_s : \mathsf{node}_s \longrightarrow \mathsf{edge}_s$*, which assigns a distinguished loop to each node;*
- *distinguished initial node* i*, and a set* O *of final nodes* o*;*

Together with the morphisms preserving all displayed structure, shapes form the category Shape*.*

Definition 3.2 *An* evolving spec*, or* **espec** *A consists of*

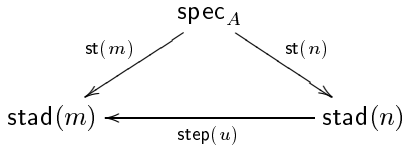
- *a spec* spec_A*, and*
- *a program* prog_A*, presented by*
  - *a shape* shape_A*;*
  - *a reflexive graph morphism* $\mathsf{st}_A : \mathsf{shape}_A \longrightarrow \mathsf{ext}_A^{op}$ *where* ext_A *is* $\mathsf{spec}_A/\mathsf{Spec}$*, the category of extensions of spec* spec_A*;*
  - *a labeling* cond *of the edges of* shape_A *by the formulas in the language of* spec_A*. That is,* st_A *maps that nodes of* shape_A *to specs in* ext_A*, and maps arcs* $u : a \to b$ *to translations in* ext_A^{op}*:* $\mathsf{st}_A(u) : st_A(a) \leftarrow st_A(b)$*. Furthemore,* st_A *maps self-loops in* shape_A *to identity translations.*

It is often convenient to also display the input and output interfaces*, presented as parameter subtheories*[1] `Xi↪stad(i)` *and* `Xo↪stad(o)`*, of the initial and the final states, respectively.*

Notation and terminology. The input and the output interfaces are usually written stad ⟨*name*⟩ init[Xi] and stad ⟨*name*⟩ fin[Xo].

If n is a node of shape_A, the codomain of $\mathsf{st}_A(n)$ is written as $\mathsf{stad}_A(n)$. If $u : m \to n$ is an edge of shape_A, its image $\mathsf{st}_A(u)$ is usually written as $\mathsf{step}_A(u)$. In summary,

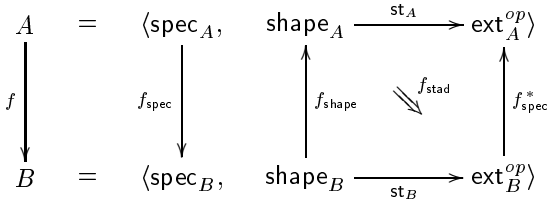
- stad assigns to each shape-node n a *state description* $\mathsf{stad}(n)$, which comes with a translation $\mathsf{st}_A(n) : \mathsf{spec}_A \longrightarrow \mathsf{stad}(n)$;
- step assigns to each shape-edge $u : m \to n$ a *step* (or *transition*) $\mathsf{step}(u) : \mathsf{stad}(m) \leftarrow \mathsf{stad}(n)$, keeping S invariant, in the sense that the following diagram commutes.



4 Refinements

We now define the concept of a refinement (or morphism) between two especs. A characteristic of espec refinements is that logical structure and behavior refine contravariantly, in opposite directions. If A refines to B, then the spec of A refines to the spec of B by a translation, but the prog of B maps into the prog of A, simulating it. So a refinement preserves the logical structure of A in B and preserves the behavior of B in A.

Definition 4.1 *Given especs* A *and* B*, a refinement* $f : A \longrightarrow B$ *consists of:*



- *a* structure map *(or translation)* f_{spec}
- *a* behavior map *(or simulation)* $f_{\mathsf{prog}} = \langle f_{\mathsf{shape}}, f_{\mathsf{stad}} \rangle$*, where*
  - f_{shape} *is a reflexive graph morphism, preserving the initial and the final nodes,*
  - f_{stad} *is* spec_A*-preserving natural transformation; this naturality and preservation amount to the commutativity of Figure 1 for every* $v : k \to \ell$ *in* shape_B *(see notes below).*

[1] By definition, the parameter $X \subseteq S$ of a parametric specification $S[X]$ can be freely instantiated, without causing any inconsistencies in the parameterized specification [7]. This also captures the idea of interface.

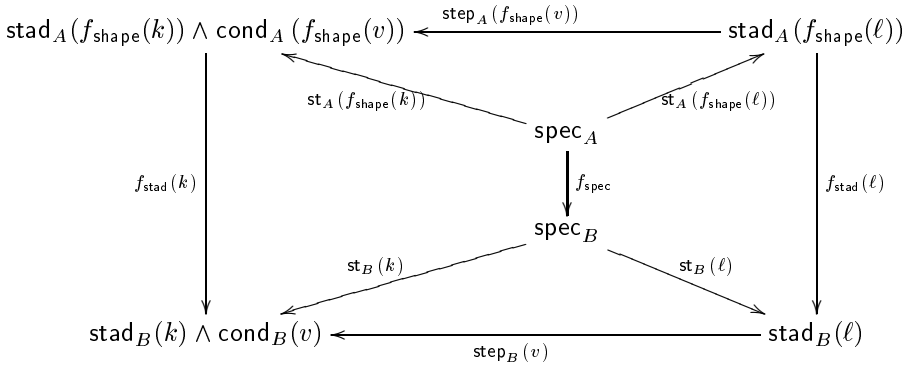


Figure 1: **Naturality Condition of a Refinement**

- *Together,* f_{spec} *and* f_{prog} *must also satisfy the* guard condition*: for every edge* $v : k \to \ell$ *in* shape_B *and edge* $u = f_{shape}(v)$ *in* shape_A

  $$\mathsf{stad}_B(k) \vdash cond_B(v) \implies f_{spec}(cond_A(u))$$

- *The inverse-image functor* f^*_{spec} *acts on the category of extensions:* $f^*_{\mathsf{spec}}(e) = e \circ f_{spec}$.

Clearly, especs and refinements form a category, which we shall denote ESpec.

Intuition. The last diagram tells that the components of f_{stad} coherently extend f_{spec} from the global specs spec_A and spec_B to their extensions stad_A and stad_B. Just like spec_B refines spec_A because it proves all formulas in the image $f_{\mathsf{spec}}[\mathsf{spec}_A]$, each $\mathsf{stad}_B(n)$ refines $\mathsf{stad}_A(f_{\mathsf{shape}}(n))$ because it proves all formulas in the image $f_{\mathsf{stad}}(n)[\mathsf{stad}_A(f_{\mathsf{shape}}(n))]$. The *structural* refinement is thus extended from $f_{\mathsf{spec}} : \mathsf{spec}_A \longrightarrow \mathsf{spec}_B$ to $f_{\mathsf{stad}} : \mathsf{stad}_A \longrightarrow \mathsf{stad}_B$. Its naturality ensures that each transition $\mathsf{step}_B(v)$ of B extends the transition $\mathsf{step}_A(f_{\mathsf{shape}}(v))$ of A.

The guard condition ensures that every behavior of B maps to a behavior of A. There are stronger versions of the guard condition that also ensure that B simulates all of A's behaviors, and others that eliminate nondeterminism. Rather than commit to one such definition, we use several, but the guard condition above is sufficient for the purposes of this paper.

Let us return to the example in Section 2. In the refinement from `GCD-0` to `GCD-1`, f_{spec} is a simple inclusion, and f_{shape} is given by the stad map

$$\begin{array}{rcl} One & \mapsto & One \\ Loop & \mapsto & One \\ Two & \mapsto & Two \end{array}$$

and the step map

$$\begin{array}{rcl} initialize & \mapsto & id_{One} \\ Loop1 & \mapsto & id_{One} \\ Loop2 & \mapsto & id_{One} \\ Out & \mapsto & Out \end{array}$$

Three of the steps map to the identity step on stad One in `GCD-0` because they only change the local variables `X` and `Y`, corresponding to identity steps in `GCD-0` (sometimes called stuttering steps). Checking the components of the natural transformation is straightforward – the proof obligations include showing that $f_{\mathsf{stad}}(k)$ is a translation for all nodes k in shape_A; e.g. that the axioms of stad `One` in `GCD-0` translate to theorems in stad `Loop` in `GCD-1`. Checking the guard condition is also straightforward; e.g. for step `Loop1` in `GCD-1`, the guard condition instantiates to

$$Loop \vdash X > Y \implies true$$

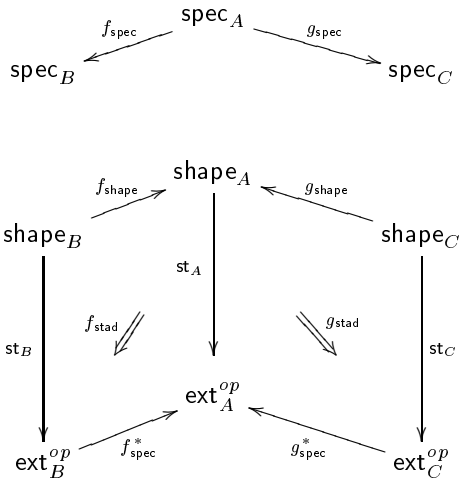
where the consequent is the guard on step id_{One} in `GCD-0`.

5 Colimits

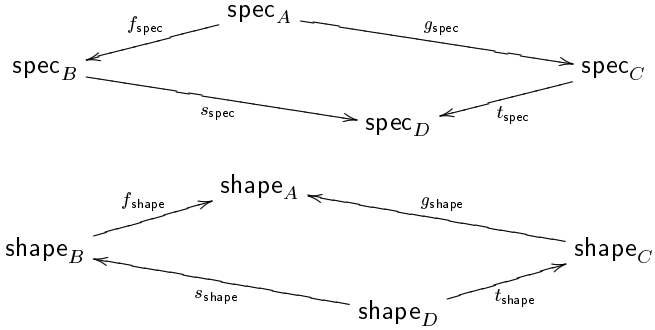
Composition of especs is carried out by the colimit operation. Colimits in ESpec are constructed from the colimits in Spec, the limits in Shape, plus some wiring to connect them in Cat. First of all, recall that all colimits can be derived from the initial object and the pushouts. Of course, the initial espec consists of the empty spec, and a one-state-one-step program (with the state represented by the empty spec).

To describe the pushout of especs, suppose we are

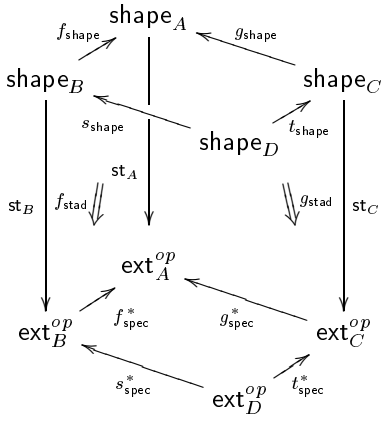
given a span of especs



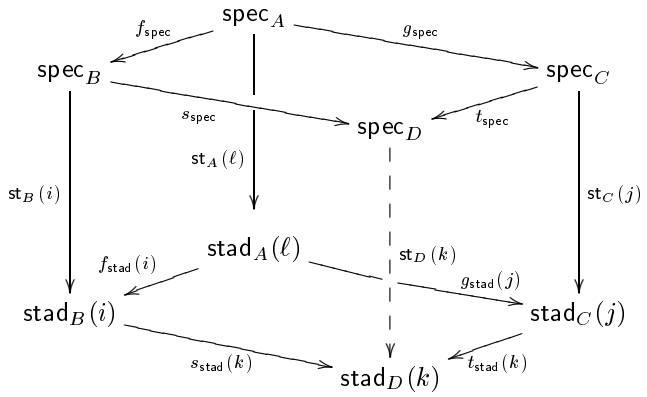
To compute the pushout, we first compute the corresponding pushout of specs and the pullback of shapes.



It is easy to see that $M : \mathsf{Spec}^{op} \longrightarrow \mathsf{Cat}$ maps the upper pushout to the pullback at the bottom of the induced cube.



If f_{stad} and g_{stad} were identities, i.e. if the two back faces of the cube were commutative, the fact that the bottom face is a pullback would induce a functor $\mathsf{st}_D : \mathsf{shape}_D \longrightarrow (\mathsf{spec}_D/\mathsf{Spec})^{op}$. Since they are not, this functor must be constructed taking f_{stad} and g_{stad} into account. The image $\mathsf{st}_D(k)$ of a node k of shape_D is now obtained as the unique arrow from the pushout at the top to the pushout at the bottom of the following cube.



Since shape_D is the pullback of f_{shape} and g_{shape}, the node k corresponds to a pair $\langle i, j\rangle$ of the nodes from shape_B and shape_C, identified in shape_A as the node $\ell = f_{\mathsf{shape}}(i) = g_{\mathsf{shape}}(j)$. Of course, $i = s_{\mathsf{shape}}(k)$ and $j = t_{\mathsf{shape}}(k)$.

This construction gives the node part stad_D of $\mathsf{st}_D : \mathsf{shape}_D \longrightarrow \mathsf{ext}_D^{op}$, as well as the components of s_{stad} and t_{stad}. The arrow part step_D is induced by the fact that the bottom of the cube is a pushout, using the naturality of f_{stad} and g_{stad}. This also yields the naturality of s_{stad} and t_{stad}. Finally we construct the guards for the edges of shape_D. Given an edge $w : k \to k'$ of shape_D define

$$\begin{aligned}\mathsf{cond}_D(w) \quad = \quad & s_{\mathsf{spec}}(\mathsf{cond}_B(s_{\mathsf{shape}}(w))) \\ & \wedge\, t_{\mathsf{spec}}(\mathsf{cond}_C(t_{\mathsf{shape}}(w)))\end{aligned}$$

A proof that this construction yields an espec with the desired universal properties may be found in [8].

Explanation. The pushout of specs is clear enough: the languages get joined together, and identified along the common part. The pullback of shapes produces the *parallel composition* of the behaviors they present. This is particularly easy to see for products, i.e. pullbacks over the final espec. For example, a product of any shape with the two-node shape $\bullet \longrightarrow \bullet$ consists of the cylinder, with the two copies of shape, and each two of their corresponding nodes connected by an edge. A product with the three-node shape $\bullet \longrightarrow \bullet \longrightarrow \bullet$ consists of three copies, similarly connected.

In general, the product of any two shapes shape_B and shape_C can be envisaged by putting a copy S_n of shape_B at each node n of shape_C, and then expanding

each edge $m \xrightarrow{u} n$ of shape_C into a cylinder from S_m to S_n, i.e. a set of parallel edges, connecting the corresponding nodes. The initial node is the pair of the initial nodes of shape_B and shape_C, whereas the final nodes are the pairs of final nodes.

In the resulting shape $\mathsf{shape}_B \times \mathsf{shape}_C$, each edge either comes from a copy of shape_B placed on a node of shape_C, or from an edge of shape_C copied to connect two particular copies of a node of shape_B; so it is either in the form $\langle$node of shape_C, edge of $\mathsf{shape}_B\rangle$, or in the form $\langle$node of shape_B, edge of $\mathsf{shape}_C\rangle$. A moment of thought shows that each path through $\mathsf{shape}_B \times \mathsf{shape}_C$ corresponds to a shuffle of a path through shape_B, and a path through shape_C; and that every such path comes about as a unique path in $\mathsf{shape}_B \times \mathsf{shape}_C$. In this sense, $\mathsf{shape}_B \times \mathsf{shape}_C$ is the *parallel composition* of shape_B and shape_C.

A pullback extracts a part of such product, identified by a pair of shape morphisms $\mathsf{shape}_B \longrightarrow \mathsf{shape}_A \longleftarrow \mathsf{shape}_C$. Since the initial node must be preserved, the initial node of the product will surely be contained in the pullback. The set of final nodes may be empty in general.

For each pair of nodes $\langle i, j\rangle$, contained in the pullback shape_D as the node k, the corresponding state description is constructed as the pushout $\mathsf{stad}_D(k)$ of $\mathsf{stad}_B(i)$ and $\mathsf{stad}_C(j)$ on the above diagram. As a theory, this state description may be inconsistent. Indeed, if B and C are not independent, but have a shared part A, their parallel composition may be globally inconsistent, in the sense that spec_D may be inconsistent; or some of the pairs of states that may come about in shuffling their computation paths may be inconsistent, which makes such paths computationally impossible. Inference tools can be used to eliminate inconsistent/unreachable stads from the colimit espec.

Despite the seeming complexity and mathematical depth in the description of the colimit, the actual computation is relatively simple. There are just three steps:

- pullback of shapes;
- pushout of specs; (the guards can be directly computed at this point)
- the pushout extensions of stads and steps.

The first two steps are simple and well known. The third one amounts to computing a pushout of theories for each stad, and using the universality of each such pushout to generate the steps from it – Epoxi's colimit algorithm returns both the cocone and a generator of translations that witness the universality of the apex.

6 Composition Example

The following example illustrates the composition of especs in the context of bank account transactions. An espec for an account deposit and an espec for an account withdrawal are composed to form an espec that *simultaneously* withdraws from one account and deposits in another. The example also indicates how especs can model some aspects of object-oriented programming. Specifically, classes are modeled as specs, and objects are classes with state. Multiple inheritance comes for free. Methods can be partially specified and refined but cannot be overridden.

```
espec Account_class is
  spec
    sort Account
    op   name    : Account -> String
    op   balance : Account -> Int
  end-spec
end-espec


espec Account_instance is
  import Account_class

  spec
;; these vars can only be changed externally
    var ext self : Account
    var ext d,n  : Int
  end-spec

  prog

;; stads can be parameterized
    stad Create[self] init[person] is
      axiom name(self)    = person
      axiom balance(self) = 0
    end-stad

    stad Account[self,x] is
      axiom balance(self) = x
    end-stad

    step Deposit[self,d]
         : Create[self] -> Account[self,d]
      balance(self) := d
    end-step
```

```
    step Change[self,n]
         : Account[self,x]
           -> Account[self,x+n]
      balance(self) := balance(self) + n
    end-step

  end-prog

end-espec
```

In order to model transfer from one account to the other, we can refine the common part of the two especs representing instances, and extend it beyond the class template, to extract the suitable transitions. The pushout of the two imports will create joint instantiation of pairs of accounts, with the parallel changes of both of them together.

```
espec Share_trans is
  import Account_class

  spec
    var ext n : Int
  end-spec

  prog

    step Change[self,n]
         : Account[self,x]
           -> Account[self,x+n]
      balance(self) |-> balance(self) + n
    end-step

  end-prog
end-espec

refinement Send
   : Share_trans -> Account_instance is
  specmap
    n |-> -n
  end-specmap
end-refinement

refinement Receive
    : Share_trans -> Account_instance
  specmap
    n |-> n
  end-specmap
end-refinement
```

The two refinements, `Send` and `Receive`, specify that the amount withdrawn is the same as the amount deposited. The pushout of `Send` and `Receive`, all under the import of `Account_class` is isomorphic to:

```
espec Transfer is import Account_class

  spec
    var ext a, b      : Account
    var ext da, db, n : Int
  end-spec

  prog

    stad Create[a,b]
         init[person_a,person_b] is
      name(a)    = person_a
      name(b)    = person_b
      balance(a) = 0
      balance(b) = 0
    end-stad

    stad Account[a,b,x,y] is
      balance(a) = x
      balance(b) = y
    end-stad

    step Deposit[a,b,da,db]
         : Create[a,b]
           -> Account[a,b,da,db]
      balance(a) := balance(a) + da
      balance(b) := balance(b) + db
    end-step

    step Change[a,b,n]
         : Account[a,b,x,y]
           -> Account[a,b,x-n,y+n]
      balance(a) := balance(a) - n
      balance(b) := balance(b) + n
    end-step

  end-prog
end-espec
```

Transitions are essentially guarded rewrites, and the transitions of the composed espec are superpositions of the transitions of the constituent machines, as noted by Fiadeiro [5] and others.

Although trivial, this example shows how the pushout of especs composes behaviors in parallel: the

transfer from one `Account_instance` to another one boils down to a parallel composition of subtracting from one account, expressed by `Send`, and adding to the other, captured by the refinement `Receive`.

7 Concluding Remarks

Epoxi builds on concepts from Specware [11], overcoming its bias towards generating functional code by supporting behavioral specifications and generation of imperative code. Epoxi also builds on previous efforts to model behavior logically (e.g. [4, 6]) by defining a formal notion of composition (via colimit) and refinement (via morphisms). Epoxi represents an advance on previous refinement methods, such as VDM and B, in a variety of ways. The categorical foundations support controlled sharing of substructure, a uniform approach to datatype refinement, and greater automated support for composition and refinement.

We are working to extend the espec formalism in several directions. First, especs naturally support the assertion of preconditions, invariants, postconditions, and safety constraints in general. However, stating liveness and fairness constraints is more difficult. Second, while a diagram-like notation is convenient for some situations, programming language notations extended with assertions may be better for other situations. It seems possible to translate from the latter back into the spec-and-translation setting for the purposes of composition and refinement. Third, in systems design it is often necessary to specify and reason about timing properties. Consequently, we are extending especs with features of timed automata [1].

In the full version of this paper [9], we show how especs support an architectural approach to system design. Architectures can be formally represented as parameterized especs where the parameter especs are the interfaces for components and connectors. The instantiation of components into the architecture is performed by refining the interface especs to the component especs and taking the colimit. The body of the architecture espec characterizes the system-level structure and invariants.

Acknowledgments: This work was supported by DARPA and the Air Force Research Lab in Rome, NY under Contract F30602-00-C-0209. Thanks to Alessandro Coglio for comments on this paper.

References

[1] Alur, R., and Dill, D. A theory of timed automata. *Theoretical Computer Science 126* (1994), 183–235.

[2] Barr, M., and Wells, C. *Category Theory for Computing Science*. Prentice-Hall, Englewood Cliffs, NJ, 1990.

[3] Errington, L. Notes on diagrams and state. Tech. rep., Kestrel Institute, 2000.

[4] Gurevich, Y. Evolving algebra 1993: Lipari guide. In *Specification and Validation Methods*, E. Boerger, Ed. Oxford University Press, 1995, pp. 9–36.

[5] J.Fiadeiro, A.Lopes, and M.Wermelinger. A mathematical semantics for architectural connectors. Tech. rep., University of Lisbon, Campo Grande, Portugal, 2001.

[6] Manna, Z., and Pnueli, A. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, 1992.

[7] 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.

[8] Pavlovic, D. Epoxi. Tech. rep., Kestrel Institute, March 2001.

[9] Pavlovic, D., and Smith, D. Composition and refinement of behavioral specifications. Tech. rep., Kestrel Institute, September 2001.

[10] Shaw, M., and Garlan, D. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, NJ, 1996.

[11] 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.