A Theory-Based Representation for Object-Oriented Domain Models

Scott A. DeLoach, *Member*, *IEEE Computer Society*, and Thomas C. Hartrum, *Member*, *IEEE*

Abstract—Formal software specification has long been touted as a way to increase the quality and reliability of software; however, it remains an intricate, manually intensive activity. An alternative to using formal specifications directly is to translate graphically based, semiformal specifications into formal specifications. However, before this translation can take place, a formal definition of basic object-oriented concepts must be found. This paper presents an algebraic model of object-orientation that defines how object-oriented concepts can be represented algebraically using an object-oriented algebraic specification language O-SLANG. O-SLANG combines basic algebraic specification constructs with category theory operations to capture internal object class structure, as well as relationships between classes.

Index Terms—Software engineering, formal methods, domain models, transformation systems.

1 INTRODUCTION

As the field of software engineering continues to evolve toward a more traditional engineering discipline, a concept that is emerging as important to this evolution is the use of *formal specifications*, the representation of software requirements by a formal language [1], [2]. Such a representation has many potential benefits, ranging from improvement of the quality of the specification itself to the automatic generation of executable code. While some impressive results have emerged from the utilization of formal specifications [3], [4], the development of formal specifications to represent a user's requirements is still a difficult task. This has restricted adoption of formal specifications by practitioners.

On the other hand, an approach to requirements modeling that *has* been gaining acceptance is the use of object-oriented methods. Initially introduced as a programming paradigm, its application has been extended to the entire software lifecycle. This *informal* approach, consisting of graphical representations and natural language descriptions, has many variations, but Rumbaugh's *Object Modeling Technique* (OMT) is typical and perhaps the most widely referenced [5]. In OMT, three models are combined to capture the essence of a software system. The *object* model captures the structural aspects of the system by defining objects, their attributes, and the relationships (associations) between them. The behavior of the system is captured by the other two models. The *dynamic* model captures the control flow as a classical state-transition model, or statechart, while the *functional* model represents the system calculations as hierarchical data flow diagrams and process descriptions. All three models are needed to capture the software system's requirements although, for a given system, one or two of the models may be of lesser importance, or even omitted.

While systems such as KIDS [3] and Specware [6] have been making progress in software synthesis, research in the acquisition of formal specifications has not been keeping pace. Formal specification of software remains an intricate, manually intensive activity. Problems associated with specification acquisition include a lack of expertise in mathematical and logical concepts among software developers, an inability to effectively communicate formal specifications with end users to validate requirements, and the tendency of formal notations to restrict solution creativity [7]. Fraser et al. suggest an approach to overcoming these problems via *parallel refinement* of semi-formal and formal specifications. In a parallel refinement approach, designers develop specifications using both semi-formal and formal representations, successively refining both representations in parallel [7].

Fig. 1 shows our concept of a parallel refinement system for formal specification development. In this system, a domain engineer would use a graphically based object-oriented interface to specify a domain model. This domain model would be automatically translated into formal Class Theories stored in a library. A user knowledgeable in the domain would then use the graphically based object-oriented interface to refine the domain model into a problem specific formal Functional Specification. Finally, a software engineer would map the Functional Specification to an appropriate formal Architecture Theory, generating a specification capable of being transformed to code by a system such as Specware.

A critical element for the success of such a system is the definition of a formal representation that captures all important aspects of object-orientation, along with a formal represention of the syntax and semantics of the informal model and a mapping for ensuring the full equivalence of the informal and formal models. While formal

• *The authors are with the US Air Force Institute of Technology, Department of Electrical and Computer Engineering AFIT/ENG, 2950 P Street, Wright-Patterson Air Force Base, OH 45433-7765.*
E-mail: {scott.deloach, thomas.hartrum}@afit.af.mil.

Manuscript received 30 Sept. 1996; revised 7 Dec. 1998; accepted 25 Feb. 1999.
Recommended for acceptance by L.K. Dillon.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 108403.

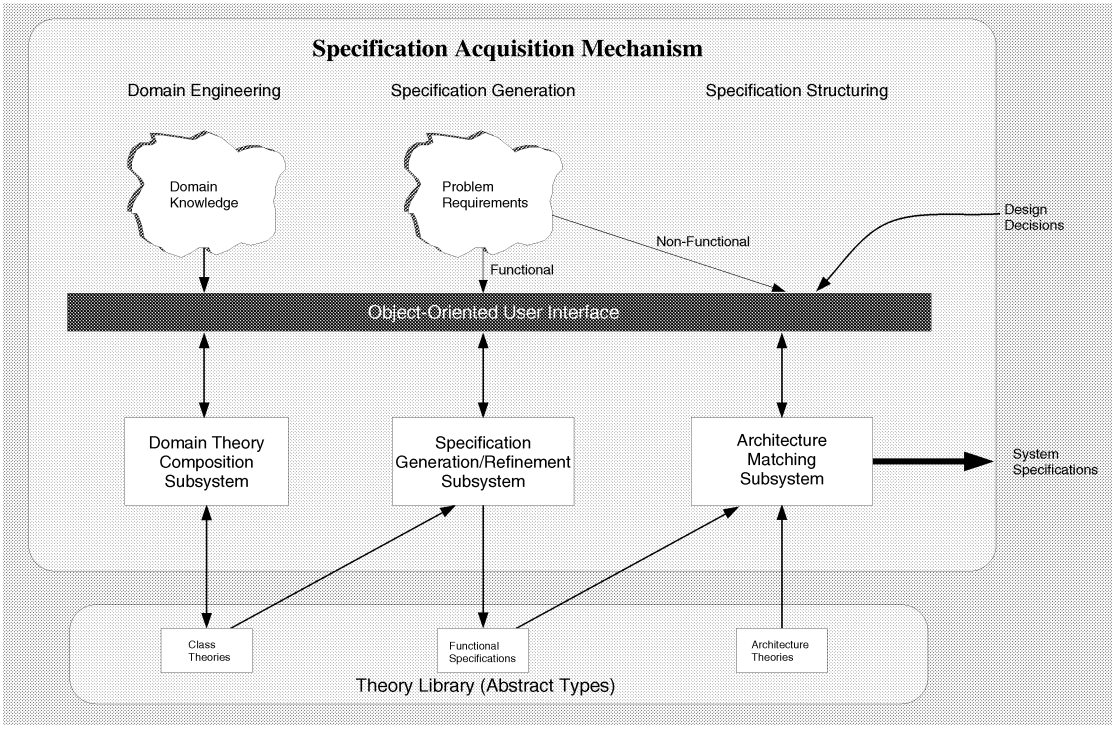


Fig. 1. Parallel refinement specification acquisition mechnanism.

representation of the informal model has been done in bits and pieces [8], [9], [10], a full, consistent, integrated formal object model does not exist. This paper describes a method for fully representing an object-oriented model using algebraic theories [11]. An algebraic language, O-SLANG, is defined as an extension of Kestrel Institute's *Slang* [12]. O-SLANG not only supports an algebraic representation of objects, but allows the use of category theory operations, such as morphisms and colimits, to combine primitive object specifications to form more complex aggregates and to extend object specifications to capture multiple inheritance [13]. Using this formal representation along with formal transformations from the informal model, we have demonstrated the automatic generation of formal algebraic specifications from commercially available object-oriented CASE tools.

The remainder of the paper is organized as follows. Section 2 discusses related work and Section 3 presents basic algebraic and category theory concepts. Section 4 introduces the basic object model, while Sections 5 through 7 describe inheritance, aggregation, and object communication in more detail. Finally, Section 8 discusses our contributions and plans for the future.

2 Related Work

There have been a number efforts designed to incorporate object-oriented concepts into formal specification languages. MooZ [14] and Object-Z [15] extend Z by adding object-oriented structures while maintaining its model-based semantics. Z++ [16] and OOZE [17] also extend Z, but provide semantics based on algebra and category theory. Although these Z extensions provide enhanced structuring techniques, they do not provide improved specification acquisition methods. FOOPS [18] is an algebraic, object-oriented specification language based on OBJ3 [19]. Both FOOPS and OBJ3 focus on prototyping and provide little support for specification acquisition. Some research has been directed toward improving specification acquisition by translating object-oriented specifications into formal specifications [10]; however, these techniques are based on Z and lack a strong notion of refinement from specification to code.

3 Theory Fundamentals

Theory-based algebraic specification is concerned with: 1) modeling system behavior using algebras (a collection of values and operations on those values) and axioms that characterize algebra behavior and 2) composition of larger specifications from smaller specifications. Composition of specifications is accomplished via specification building operations defined by category theory constructs [20]. A *theory* is the set of all assertions that can be logically proven from the axioms of a given specification. Thus, a specification defines a theory and is termed a *theory presentation*.

In algebraic specifications, the structure of a specification is defined in terms of *sorts*, abstract collections of values, and *operations* over those sorts. This structure is called a *signature*. A signature describes the structure of a solution; however, a signature does not specify semantics. To specify semantics, the definition of a signature is extended with axioms defining the intended semantics of signature operations. A signature with associated axioms is called a *specification*. An example of a specification is shown in Fig. 2.

A specification allows us to formally define the internal structure of object classes (attributes and operations); however, it does not provide the capability of reasoning

```
spec ARRAY is
sorts E, I, A
operations
    assign  : A, I, E → A
    apply   : A, I → E
axioms ∀ (i,j ∈ I, a ∈ A, e ∈ E)
    (i = j) ⇒ apply(assign(a,i,e),j) = e;
    (i ≠ j) ⇒ apply(assign(a,i,e),j) = apply(a,j)
end
```

Fig. 2. Array specification.

about relationships between object classes. To create theory-based algebraic specifications that parallel object-oriented specifications, the ability to define and reason about relationships between theories, similar to those used in object-oriented approaches (inheritance, aggregation, etc.), must be available. Category theory is an abstract mathematical theory used to describe the external structure of various mathematical systems [21] and is used here to describe relationships between specifications.

A category consists of a collection of *C-objects* and *C-arrows* between objects such that: 1) there is a C-arrow from each object to itself, 2) C-arrows are composable, and 3) arrow composition is associative. An obvious example is the category **Set**, where "C-objects" are sets and "C-arrows" are functions between sets. However, of greater interest in our research is the category **Spec**. **Spec** consists of specifications as the "C-objects" with specification morphisms as the "C-arrows." A *specification morphism*, σ, is a pair of functions that map sorts (σ_S) and operations (σ_Ω) from one specification to compatible sorts and operations of a second specification such that the axioms of the first specification are theorems of the second specification. Intuitively, specification morphisms define how one specification is embedded in another. An example of a morphism from *array* to *finite-map* (Fig. 3) is shown below.

$$\sigma_\Omega = \{\text{assign} \mapsto \text{update},\ \text{apply} \mapsto \text{apply}\}$$
$$\sigma_S = \{\text{A} \mapsto \text{M},\ \text{I} \mapsto \text{D},\ \text{E} \mapsto \text{R}\}$$

Specification morphisms comprise the basic tool for defining and refining specifications. Our toolset can be extended to allow the creation of new specifications from a set of existing specifications. Often two specifications derived from a common ancestor specification need to be combined. The desired combination consists of the unique parts of two specifications and some "shared part" common to both specifications (the part defined in the shared ancestor specification). This combining operation is a *colimit*.

Conceptually, the colimit is the "shared union" of a set of specifications based on the morphisms between the specifications. These morphisms define equivalence classes of sorts and operations. For example, if a morphism, σ, from specification A to specification B maps sort α to sort β, then α and β are in the same equivalence class and thus become a single sort in the colimit specification of A, B, and σ. The colimit operation creates a new specification, the *colimit specification*, and a specification morphism from each specification to the colimit specification. An example showing the relationship between a colimit and multiple inheritance is provided in Section 5.

```
spec FINITE-MAP is
sorts M, D, R
operations
  empty   : → M
  update  : M, D, R → M
  apply   : A, D → R
  def?    : A, D → Boolean
axioms ∀ (d1,d2 ∈ D, m ∈ M, r ∈ R)
  (d1 = d2) ⇒ apply(update(m,d2,r),d1) = r;
  (d1 ≠ d2) ⇒ apply(assign(m,d2,r),d1) = apply(m,d1);
  def?(update(m,d2,r),d1) = (d1 = d2) ∨ def?(m,d1);
  def?(empty,d1) = false
end
```

Fig. 3. Finite map specification.

From these basic tools (morphisms and colimits), we can construct specifications in a number of ways [20]. We can, 1) build a specification from a signature and a set of axioms, 2) form the union of a set of specifications via a colimit, 3) rename sorts or operations via a specification morphism, and 4) parameterize specifications. Many of these methods are useful in translating object-oriented specifications into theory-based specifications. Detailed semantics of object-oriented concepts using specifications and category theory constructs are presented next.

4 Object Classes

The building block of object-orientation is the object *class* which defines the structure of an object and its response to external stimuli based its current state. Formally defined in Section 4.1 as a *class type*, a class is a template from which individual object *instances* can be created. Fig. 4 shows a specification of a banking account class in O-SLANG.

4.1 Class Structure

In our theory-based object model, we capture the structure of a class as a theory presentation, or algebraic specification, as follows.

Definition 1. Object Class Type *A class type, C, is a signature, $\Sigma = \langle S, \Omega \rangle$ and a set of axioms, Φ, over Σ (i.e., a theory presentation, or specification) where*

- S *denotes a set of sorts including the class sort*
- Ω *denotes a set of functions over S*
- Φ *denotes a set of axioms over Σ.*

Sorts in S are used to describe collections of data values used in the specification. In O-SLANG, a distinguished sort, the *class sort*, is the set of all possible objects in the class. In an algebraic sense, this is really the set of all possible abstract value representations of objects in the class. Functions in Ω are classified in O-SLANG syntax as attributes, methods, state-attributes, states, events, and operations. *Attributes* are defined implicitly by visible

functions which return specific data values. In Fig. 4, the functions *date* and *bal* are attributes. *Methods* are nonvisible functions invoked *via* visible events that modify an object's attribute values. A method's domain includes an object, along with additional parameters, while the return value is always the modified object. In Fig. 4, the functions *create-acct*, *credit*, and *debit* are methods. The semantics of functions, as well as invariants between class attribute values, are defined using first order predicate logic *axioms*. In general, axioms define methods by describing their effects on attribute values as in the following example.

$$\forall(a : Acct, x : Amnt)\ \ bal(credit(a, x)) = bal(a) + x;$$

4.2 Class Behavior

4.2.1 States

In our model, a *state* is a partition of the cross-product of an object's attribute values. For example, a bank account might be partitioned into an *ok* and an *overdrawn* state based on a partitioning of its balance values. Formally, a class type has at least one *state sort* (multiple state sorts allow modeling concurrent state models and substate models), a set of *states* which are elements in a state sort (defined by nullary functions), a *state attribute* defined over each state sort as a function which returns the current state of an object, and a set of *state invariants*, axioms that describe constraints on class attributes that must hold true while in a given state. In our object model, we separate state attributes from normal attributes to capture the notion of an object's abstract state as might be defined in an statechart. The values of state attributes of define an object's *abstract* state while the values of normal attributes define an object's true state. In Fig. 4, the class state sort is *Acct-State*, the class state attribute is *acct-state*, the state constants are *ok* and *overdrawn*, and the state invariants are

$$acct\text{-}state(a) = ok \Rightarrow bal(a) \geq 0;$$
$$acct\text{-}state(a) = overdrawn \Rightarrow bal(a) < 0;$$

These axioms state that, when the balance of an account is greater than or equal to zero, the account must be in the *ok* state; however, when the balance of the account becomes less than zero, the state must become *overdrawn*. While it is tempting to replace the implication operators with equivalence operators, doing so would unnecessarily restrict subclasses derived from this class as defined in Section 5. Additionally, the axiom

$$ok \neq overdrawn$$

ensures correct interpretation of the specification that states *ok* and *overdrawn* are distinct.

4.2.2 Events

Events are visible functions that allow objects to communicate with each other and may directly modify state attributes. We present a more detailed discussion of the specification of this communication between objects in Section 7. As a side effect, receipt of an event may cause the invocation of methods or the generation of events sent to other objects. Events are distinct from methods to separate control from execution. This separation keeps us from having to embed state-based control information within methods. Each class has a *new* event which triggers the *create* method and initializes the object's state attributes. In Fig. 4, the functions *new-acct*, *deposit*, and *withdrawal* are events. The effect of these events on the class behavior, which can be represented by the statechart in Fig. 5, is defined by a set of axioms similar to the following Fig. 4.

$$\begin{aligned} acct\text{-}state(a) = ok \wedge bal(a) < x \\ \Rightarrow acct\text{-}state(withdrawal(a, x)) \\ = overdrawn. \end{aligned}$$

4.2.3 Class Operations

Operations are visible functions that are generally used to compute derived attributes and may not directly modify attribute values. In Fig. 4, the function *acct-attr-equal* is an operation. Similar to methods and events, the semantics of operations are defined using first order predicate logic axioms.

5 Inheritance

Class inheritance plays an important role in object-orientation; however, the correct use of inheritance is not uniformly agreed upon. In our work, we have chosen to use a strict form of inheritance that allows a subclass object to be freely substituted for its superclass in any situation. This subtype interpretation was selected to simplify reasoning about the class's properties and to keep it closely related to software synthesis concepts [6]. We believe the advantages of strict inheritance outweigh its disadvantages in our research since most arguments favoring a less strict approach to inheritance—such as polymorphism and overloading—are much more germane to implementation than to specification. Thus, as a subtype, a subclass may only *extend* the features of its superclass. Liskov defines these desired effects as the "substitution property" [22]:

> If for each object o_1 of type S, there is an object o_2 of type T such that, for all programs P defined in terms of T, the behavior of P is unchanged when o_1 is substituted for o_2, then S is a subtype of T.

The only way to ensure the substitution property holds in all cases is to ensure that the effects of all superclass operations performed on an object are equivalent in the subclass and the superclass. To achieve this, inheritance must provide a mapping from the sorts, operations, and attributes in the superclass to those in the subclass that preserve the semantics of the superclass. This is the basic definition of a specification morphism (extended for O-SLANG to map class-sorts to class-sorts, attributes to attributes, methods to methods, etc.) and provides us a formal definition of inheritance [13].

Specification morphisms map the sorts and operations of one algebraic specification into the sorts and operations of a second specification such that the axioms in the first specification are theorems in the second specification [13]. Thus, in essence, a specification morphism defines an

class ACCT is
import Amnt, Date
class sort Acct
sorts Acct-State
operations
acct-attr-equal : Acct, Acct → Boolean
attributes
date : Acct → Date
bal : Acct → Amnt
state-attributes
acct-state : Acct → Acct-State
methods
create-acct : Date → Acct
credit, debit : Acct, Amnt → Acct
states
ok, overdrawn : → Acct-State
events
new-acct : Date → Acct
deposit, withdrawal : Acct, Amnt → Acct
axioms
% *state uniqueness and invariant axioms*
ok ≠ overdrawn;
∀ (a: Acct) acct-state(a) = ok ⇒ bal(a) ≥ 0;
∀ (a: Acct) acct-state(a) = overdrawn ⇒ bal(a) < 0;
% *operation definitions*
∀ (a,a1: Acct) acct-attr-equal(a, a1) ⇔ date(a) = date(a1) ∧ bal(a) = bal(a1);
% *method definitions*
∀ (d: Date) date(create-acct(d)) = d ∧ bal(create-acct(d)) = 0;
∀ (a: Acct, x: Amnt) bal(credit(a,x)) = bal(a) + x ∧ date(credit(a,x)) = date(a)
% *event definitions*
∀ (d: Date) acct-state(new-acct(d))=ok ∧ acct-attr-equal(new-acct(d), create-acct(d))
∀ (a: Acct, x: Amnt) acct-state(a)=ok
⇒ acct-state(deposit(a,x))=ok ∧ acct-attr-equal(deposit(a,x), credit(a,x));
∀ (a: Acct, x: Amnt) acct-state(a)=overdrawn ∧ bal(a) + x ≥ 0
⇒ acct-state(deposit(a,x))=ok ∧ acct-attr-equal(deposit(a,x), credit(a,x));
∀ (a: Acct, x: Amnt) acct-state(a)=overdrawn ∧ bal(a) + x < 0
⇒ acct-state(deposit(a,x))=overdrawn ∧ acct-attr-equal(deposit(a,x), credit(a,x));
∀ (a: Acct, x: Amnt) acct-state(a)=ok ∧ bal(a) ≥ x
⇒ acct-state(withdrawal(a,x))=ok ∧ acct-attr-equal(withdrawal(a,x), debit(a,x));
∀ (a: Acct, x: Amnt) acct-state(a)=ok ∧ bal(a) < x
⇒ acct-state(withdrawal(a,x))=overdrawn ∧ acct-attr-equal(withdrawal(a,x), debit(a,x));
∀ (a: Acct, x: Amnt) acct-state(a)=overdrawn
⇒ acct-state(withdrawal(a,x))=overdrawn ∧ acct-attr-equal(withdrawal(a,x), a)
end-class

Fig. 4. Object class.

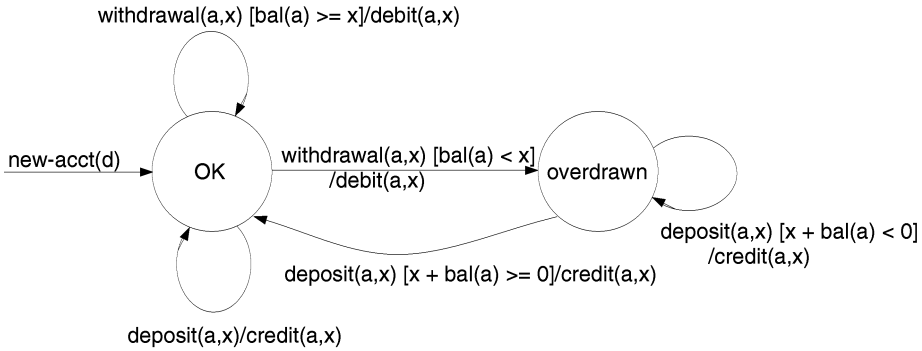


Fig. 5. Account statechart.

```
class SAcct is
import Acct, Rate
class sort SAcct < Acct
operations
  sacct-attr-equal : SAcct, SAcct → Boolean
attributes
  rate : SAcct → Rate
  int-date : SAcct → Date
methods
  create-sacct : Date → SAcct
  set-rate : SAcct, Date, Rate → SAcct
  comp-int : SAcct, Date → SAcct
events
  new-sacct : Date → SAcct
  rate-change : SAcct, Date, Rate → SAcct
  compute-interest : SAcct, Date → SAcct
axioms ∀ (d: Date, r: Rate, a, a1: SAcct)
 % operation definitions
  ∀ (a,a1: SAcct) sacct-attr-equal(a, a1) ⇔ rate(a) = rate(a1) ∧ int-date(a) = int-date(a1)
    ∧ acct-attr-equal(a, a1);
 % method definition
  ∀ (d: Date) date(create-sacct(d)) = date(create-acct(d)) ∧ bal(create-sacct(d)) = bal(create-acct(d))
    ∧ acct-state(create-sacct(d)) = acct-state(create-acct(d)) ∧ int-date(create-sacct(d)) = d
    ∧ rate(create-sacct(d)) = 0;
  ∀ (s: SAcct, a: Amnt) rate(credit(s,a)) = rate(s) ∧ int-date(credit(s,a)) = int-date(s);
  ∀ (s: SAcct, a: Amnt) rate(debit(s,a)) = rate(s) ∧ int-date(debit(s,a)) = int-date(s);
  ∀ (d: Date, r: Rate, a: SAcct) rate(set-rate(a,d,r)) = r ∧ int-date(set-rate(a,d,r)) = d
    ∧ bal(set-rate(a,d,r)) = bal(a) ∧ date(set-rate(a,d,r)) = date(a);
  ∀ (d: Date, a: SAcct) rate(comp-int(a,d)) = rate(a) ∧ int-date(comp-int(a,d)) = d
    ∧ bal(a) ≥ 0 ⇒ bal(comp-int(a,d)) = bal(a) + rate(a) * ((d - int-date(a))/days-per-year(d))
    ∧ bal(a) ≤ 0 ⇒ bal(comp-int(a,d)) = bal(a) ∧ date(comp-int(a,d)) = date(a);
 % event definitions
  ∀ (d: Date) acct-state(new-sacct(d)) = ok ∧ sacct-attr-equal(new-sacct(d), create-sacct(d));
  ∀ (d: Date, r: Rate, a: SAcct) acct-state(a) = ok ⇒ acct-state(rate-change(a,d,r)) = ok
    ∧ sacct-attr-equal(rate-change(a,d,r),set-rate(comp-int(a,d),d,r));
  ∀ (d: Date, r: Rate, a: SAcct) acct-state(a) = overdrawn
    ⇒ acct-state(rate-change(a,d,r)) = overdrawn
      ∧ sacct-attr-equal(rate-change(a,d,r),set-rate(comp-int(a,d),d,r));
  ∀ (d: Date, a: SAcct) acct-state(a) = ok ⇒ acct-state(compute-interest(a,d)) = ok
    ∧ sacct-attr-equal(compute-interest(a,d),comp-int(a,d));
  ∀ (d: Date, a: SAcct) acct-state(a) = overdrawn ⇒ acct-state(compute-interest(a,d)) = overdrawn
    ∧ sacct-attr-equal(compute-interest(a,d),a)
end-class
```

Fig. 6. Savings class.

embedding of functionality from one specification into a second specification.

Definition 2. Inheritance *A class D is said to inherit from a class C, denoted $D < C$, if there exists a specification morphism from C to D and the class sort of D is a subsort of the class sort of C (i.e., $D_{cs} \subseteq C_{cs}$).*

Definition 2 provides a concise, mathematically precise definition of inheritance and ensures the preservation of the substitution property as stated in Theorem 1 [11].

Theorem 1. *Given a specification morphism, $\sigma : C \rightarrow D$, between two internally consistent classes C and D such that $D_{cs} \subseteq C_{cs}$, the substitution property holds between C and D.*

Since we assume user defined specifications are initially consistent, we can ensure consistency in a subclass as long as the user does not introduce new axioms in the subclass that redefine how a method defined in the superclass affects an attribute also defined in the superclass.

An example of single inheritance using a subclass of the *ACCT* class is shown in Fig. 6. The *import* statement includes all the sorts, functions, and axioms declared in the *ACCT* class directly into the new class, while the class sort declaration $SAcct < Acct$ states that *SAcct* is a subsort of *Acct* and, as such, all functions and axioms that apply to an *Acct* object apply to a *SAcct* object as well. A statechart for *SACCT* is shown in Fig. 7. The import operation defines a specification morphism between *ACCT* and *SACCT*, while the subsort declaration completes the requirements of Definition 2 for inheritance. Therefore, *SACCT* is a valid subclass of *ACCT* and the substitution property holds.

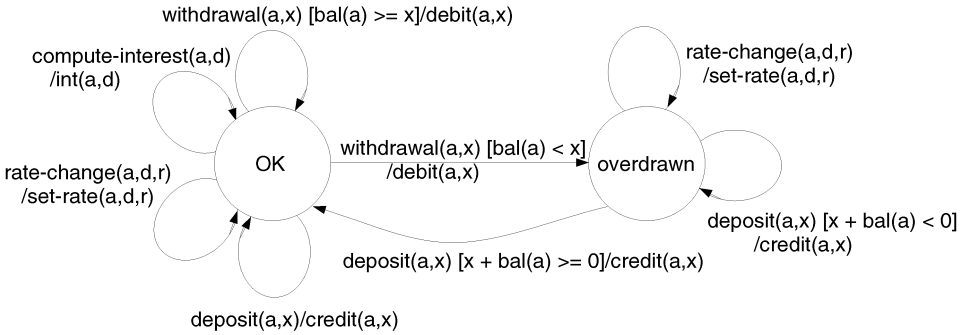


Fig. 7. Savings account statechart.

5.1 Multiple Inheritance

Multiple inheritance requires a slight modification to the notion of inheritance as stated in Definition 2. The set of superclasses must first be combined via a category theory colimit operation and then used to "inherit from."

Based on specification morphisms, the *colimit* operation is composed of a set of existing specifications to create a new *colimit* specification [21]. This new *colimit* specification contains all the sorts and functions of the original set of specifications without duplicating the "shared" sorts and functions from a common "ancestor" specification. Conceptually, the colimit of a set of specifications is the "shared union" of those specifications. Therefore, the colimit operation creates a new specification, the *colimit specification*, and a morphism from each specification to the colimit specification.

Definition 3. Multiple Inheritance *A class D multiply inherits from a set of classes $\{C_1 .. C_n\}$ if there exists a specification morphism from the colimit of $\{C_1 .. C_n\}$ to D such that the class sort of D is a subsort of each of the class sorts of $\{C_1 .. C_n\}$.*

This definition states that all sorts and operations from each superclass map to sorts and operations in the subclass such that the defining axioms are logical consequences of the axioms of the subclass. This implies that all operations defined in superclasses are applicable in the subclass as well. This definition ensures that the subclass D inherits, in the sense of Definition 2, from each superclass in $\{C_1 .. C_n\}$, as shown in Theorem 2 below [11].

Theorem 2. *Given a specification morphism from the colimit of $\{C_1 .. C_n\}$ to D such that the class sort of D is a subsort of each of the class sorts of $\{C_1 .. C_n\}$, the substitution property holds between D and each of its superclasses $\{C_1 .. C_n\}$*

It is important to note that Definition 3 only ensures valid inheritance when the axioms defining each operation in the superclass specifications $\{C_1 .. C_n\}$ are complete. Failure to completely define operations can result in inconsistent colimit specifications [11].

We can use multiple inheritance to combine the features of a savings account with those of a checking account, *CACCT*, as defined in Fig. 8. To compute the resulting class, the colimit of the classes *ACCT*, *SACCT*, *CACCT*, and morphisms from *ACCT* to *SACCT* and *CACCT* is computed, as shown in Fig. 9, where an arrow labeled with an "i" represents an import morphism and a "c" represents a morphism formed by the colimit operation. A simple extension of the colimit specification with the class sort definition

$$Comb\text{-}Acct < SAcct, CAcct$$

yields the desired class, where *Comb-Acct* is a subclass of both *SAcct* and *CAcct*, as denoted by the $<$ operator in the class sort definition. Fig. 10 shows the "long" version of the combined specification signature with all the attributes,

```
class CACCT is
import Acct class sort CAcct < Acct
operations
   cacct-attr-equal : CAcct, CAcct → Boolean
attributes
   check-cost : CAcct → Amnt
methods
   create-cacct : Date → CAcct
   set-check-cost : CAcct, Amnt → CAcct
events
   new-cacct : Date → CAcct
   change-check-cost : CAcct, Amnt → CAcct
   write-check : CAcct, Amnt → CAcct
axioms ∀ (a: CAcct, x: Amnt)
  % axioms omitted
end-class
```

Fig. 8. Checking class.

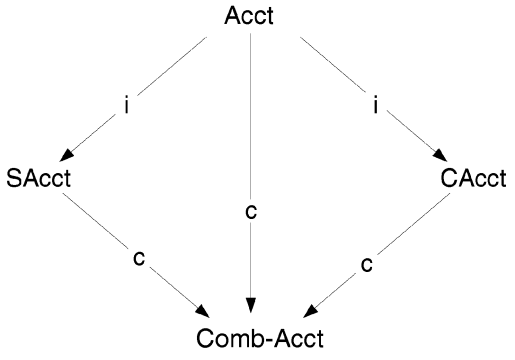


Fig. 9. Colimit of accounts.

```
class COMB-ACCT is
import SAcct, CAcct
class sort Comb-Acct < SAcct, CAcct
sorts Acct-State
operations
  comb-acct-attr-equal : Comb-Acct, Comb-Acct → Boolean
attributes
  date : Comb-Acct → Date
  bal : Comb-Acct → Amnt
  rate : Comb-Acct → Rate
  int-date : Comb-Acct → Date
  check-cost : Comb-Acct → Amnt
state-attributes
  acct-state : Comb-Acct → Acct-State
methods
  create-acct : Date → Comb-Acct
  create-sacct : Date → Comb-Acct
  create-cacct : Date → Comb-Acct
  create-comb-acct : Date → Comb-Acct
  credit : Comb-Acct, Amnt → Comb-Acct
  debit : Comb-Acct, Amnt → Comb-Acct
  set-rate : Comb-Acct, Date, Rate → Comb-Acct
  int : Comb-Acct, Date → Comb-Acct
  set-check-cost : Comb-Acct, Amnt → Comb-Acct
  write-check : Comb-Acct, Amnt → Comb-Acct
states
  ok : → Acct-State
  overdrawn : → Acct-State
events
  new-acct : Date → Comb-Acct
  new-sacct : Date → Comb-Acct
  new-cacct : Date → Comb-Acct
  new-comb-acct : Date → Comb-Acct
  deposit : Comb-Acct, Amnt → Comb-Acct
  withdrawal : Comb-Acct, Amnt → Comb-Acct
  rate-change : Comb-Acct, Date, Rate → Comb-Acct
  compute-interest : Comb-Acct, Date → Comb-Acct
  change-check-cost : Comb-Acct, Amnt → Comb-Acct
  write-check : Comb-Acct, Amnt → Comb-Acct
axioms ∀ (a: CAcct, x: Amnt)
  % axioms omitted
```

Fig. 10. Combined account signature.

methods, and events inherited by the *Comb-Acct* class (axioms are omitted for brevity).

6 AGGREGATION

Aggregation is a relationship between two classes where one class, the *aggregate*, represents an entire assembly and the other class, the *component*, is "part-of" the assembly. Not only do aggregate classes allow the modeling of systems from components, but they also provide a convenient context in which to define constraints and associations between components. Aggregate class behavior is defined by that of its components and the constraints between them. Thus, aggregates impose an *architecture* on the domain model and specifications derived from it.

Components of an aggregate class are modeled similarly to attributes of a class through the concept of *Object-Valued attributes*. An object-valued attribute is a class attribute whose sort type is a set of objects—the class-sort of another class. Formally, they are specification functions that take an object and return an external object or set of objects. The effects of methods on object-valued attributes are similar to those for normal attributes. However, instead of directly specifying a new value for an object-valued attribute, an event is sent to the object stored in the object-valued attribute. We can formally define an aggregate using the colimit operation and object-valued attributes.

Definition 4. Aggregate *A class C is an aggregate of a set of component classes, $\{D_1..D_n\}$, if there exists a specification*

```
class ACCT-CLASS is
contained-class ACCT
class sort Acct-Class
events
   new-acct-class : → Acct-Class
   withdrawal : Acct-Class, Amnt → Acct-Class
   deposit : Acct-Class, Amnt → Acct-Class
axioms
   new-acct-class() = empty-set;
   ∀ (a: Acct, ac: Acct-Class, x: Amnt) a ∈ ac ⇔ deposit(a,x) ∈ deposit(ac,x);
   ∀ (a: Acct, ac: Acct-Class, x: Amnt) a ∈ ac ⇔ withdrawal(a,x) ∈ withdrawal(ac,x)
end-class
```

Fig. 11. O-SLANG class set specification.

morphism from the colimit of $\{D_1..D_n\}$ to C such that C has at least one corresponding object-valued attribute for each class sort in $\{D_1..D_n\}$.

An aggregate class combines a number of classes via the colimit operation to specify a system or subsystem. The colimit operation also unifies sorts and functions defined in separate classes and associations to ensure that the associations actually relate two (or more) specific classes. To capture a domain model within a single structure, we can create a domain-level aggregate. To create this aggregate, the colimit of all classes and associations within the domain is taken.

6.1 Aggregate Structure

An aggregate consists of a number of classes and provides a convenient means to define additional constructs and relationships. These constructs include class sets, individual components, and associations.

6.1.1 Class Set

A class type definition specifies a template for creating new instances. In order to manage a set of objects in a class, a *class set* is created for each class defined.

Definition 5. Class Set *A class set is a class whose class sort is a set of objects from a previously defined object class, C. A class set includes a "class event" definition for each event in C such that the reception of a class event by a class set object sends the corresponding event in C to each object of type C contained in the class set object. If the class C is a subclass of $D_1...D_n$, then the class set of C is a subclass of the class sets of $D_1...D_n$.*

The class set creates a class type whose class sort is a set of objects and defines some basic functions on that set. For example, in Fig. 11, *ACCT-CLASS* imports the *ACCT* class specification and adds additional "class" events. These class events mirror the "object" events defined in the class type and distribute the event invocation to each object in the class set. The resulting specification is effectively a *set* of *Acct* objects. Using the category theory colimit operation, a class type specification can be combined with a basic *SET* specification to automatically derive the class set specification.

6.1.2 Specification of Components

Components may have either a fixed, variable, or recursive structure. All three structures use object-valued attributes to reference other objects and define the aggregate. The difference between them lies in the types of objects that are referenced and the functions and axioms defined over object-valued attributes. In a fixed configuration, once an aggregate references a particular object, that reference may not be changed. The ability of an aggregate object to change the object references of its object-valued attributes is determined by whether a method exists, other than the initialization method, to modify the object-valued attribute. If no methods modify any object-valued attributes then the aggregate is fixed. If methods do modify the object-valued attributes, then the aggregate is variable. A recursive structure is also easily represented using object-valued attributes. In this case, an object-valued attribute is defined in the class type that references its own class sort.

6.1.3 Associations

Associations model the relationships between an aggregate's components. We define a *link* as a single connection between object instances and an *association* as a group of such links. A link defines what object classes may be connected along with any attributes or functions defined over the link. *Link attributes* and *link functions* are those that do not belong to any one of the objects involved, but exist only when there is a link between objects. Formally, associations are represented generically as a specification that defines a set of individual links. A link defines a specification that uses object-valued attributes to reference individual objects from two or more classes. Links may also define link attributes or functions in a manner identical to object classes. Basically, a link is a class whose class-set is an association while an association is a set of links. Associations with more than two classes are handled in a similar manner by simply adding additional object-valued attributes.

$$
\begin{array}{ll}
\text{exactly one} & \mapsto \text{size(image(a,o))} = 1 \\
\text{many} & \mapsto \text{size(image(a,o))} \geq 0 \\
\text{optional} & \mapsto \text{size(image(a,o))} = 1 \vee \text{size(image(a,o))} = 0 \\
\text{one or more} & \mapsto \text{size(image(a,o))} \geq 1 \\
\text{numerically specified} & \mapsto \text{size(image(a,o))} = x \\
\text{numerically specified} & \mapsto \text{size(image(a,o))} \geq x \wedge \text{size(image(a,o))} \leq y
\end{array}
$$

Fig. 12. Association multiplicity axioms.

Definition 6. Link *A link is an object class type with two or more object-valued attributes.*

Definition 7. Association *An association is the class set of a link specification.*

Multiplicity is defined as the number of links of an association in which any given object may participate. For a binary association, an *image* operation is defined for each class in the association. The image operation returns a set of objects with which a particular object is associated and is used to define multiplicity constraints, as shown in Fig. 12. For binary associations, we allow five categories of association multiplicities: exactly one, many, optional, one or more, or numerically specified. True ternary or higher level associations are relatively rare; however, they can be modeled using an association class. In a ternary association, the image operation returns a set of object tuples associated with a given object. Since the output is a set of tuples, the same multiplicity axioms, as shown in Fig. 12, apply.

6.1.4 Banking Example

An example of a link specification between a class of customers *CUST* (not illustrated) and the *ACCT* class to associate customers with their accounts shown in Fig. 13. The *CA-Link* link specification can relate objects from the two classes without embedding internal references into the classes themselves. Although the names of the object-valued attributes and sorts correspond to the *CUSTOMER* and *ACCT* classes, the link specification does not formally tie the classes together. This relationship is actually formalized in the aggregate specification. The association between the *ACCT* class and the *CUSTOMER* class is shown in Fig. 14.

The *CUST-ACCT* class defines a set of *CA-Link* objects, while its axioms define the multiplicity relationships between accounts and customers, in this case, exactly one

```
link CA-LINK is
class sort CA-Link
sorts Customer, Account
operations
  ca-link-attr-equal : CA-Link, CA-Link → Boolean
attributes
  customer : CA-Link → Customer
  account : CA-Link → Account
methods
  create-ca-link : Customer, Account → CA-Link
events
  new-ca-link : Customer, Account → CA-Link
axioms
  % operation definition
  ∀ (c,c1: Customer)
    ca-link-attr-equal(c,c1) ⇔ customer(c) = customer(c1) ∧ account(c) = account(c1);
  % create method definition
  ∀ (c: Customer, a: Account)
    customer(create-ca-link(c,a)) = c ∧ account(create-ca-link(c,a)) = a;
  % new event definition
  ∀ (c: Customer, a: Account)
    ca-link-attr-equal(new-ca-link(c,a), create-ca-link(c,a))
end-link
```

Fig. 13. Customer account link.

```
association CUST-ACCT is
link-class CA-Link
class sort Cust-Acct
sorts Accounts, Customers
methods
  image : Cust-Acct, Customer → Accounts
  image : Cust-Acct, Account → Customers
events
  new-cust-acct : → Cust-Acct
axioms
 % multiplicity axioms
  ∀ (ca: Cust-Acct, c: Customer) size(image(ca, c)) ≥ 1;
  ∀ (ca: Cust-Acct, a: Account) size(image(ca, a)) = 1;
 % new event definition
  new-cust-acct() = empty-set;
  ... definition of image operations ...
end-association
```

Fig. 14. Cust-Acct association.

customer per account, while each customer may have one or more accounts.

The *CUSTOMER*, *ACCT*, and *CUST-ACCT* classes are then combined to form an, aggregate BANK. The sorts from *CUST* and *CUST-ACCT* and the sorts from *ACCT* and *CUST-ACCT* are unified via specification morphisms that define their equivalence, as shown in Fig. 15. The actual specification of the aggregation colimit *BANK-AGGREGATE* is not shown, but is further refined into the aggregate specification for *BANK*, as shown in Fig. 16. The *SET* specification is used to unify sorts, while the *INTEGER* specification ensures only a single copy of integers is included. Three copies of the *SET* specification are included since each class requires a unique set.

Once the *BANK-AGGREGATE* specification is computed, the *CUST-ACCT* association actually associates the *CUSTOMER* class to the *ACCT* class. New functions and axioms can be added to an extension of colimit specification, the *BANK* class type specification, as shown in Fig. 16, to describe aggregate-level interfaces and aggregate behavior based on component events and methods.

6.2 Aggregate Behavior

Once an aggregate is created via a colimit operation, further specification is required to make the aggregate behave in an integrated manner. First, new aggregate level functions are defined to enable the aggregate to

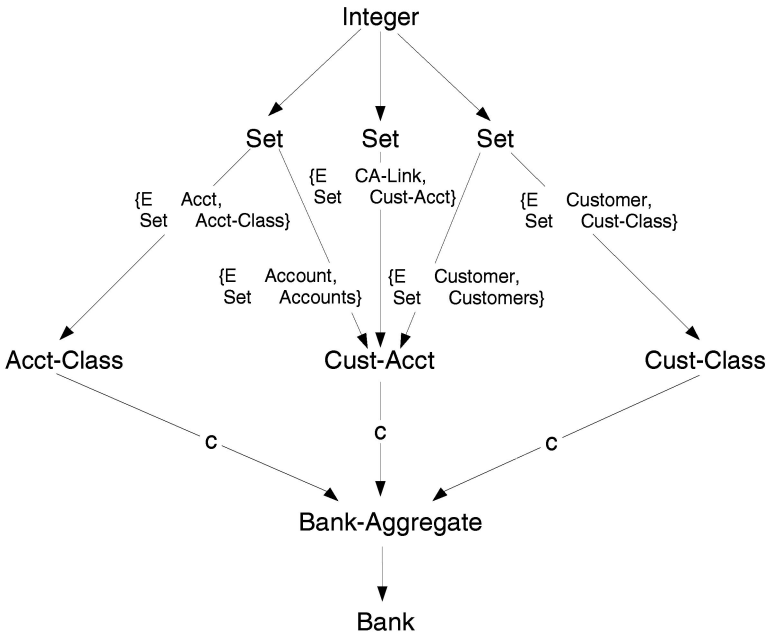


Fig. 15. Aggregation composition.

```
class BANK is
import BANK-AGGREGATE
class sort Bank
attributes
  acct-obj : Bank → Acct-Class
  cust-obj : Bank → Cust-Class
  cust-acct-assoc : Bank → Cust-Acct
methods
  aggregate methods defined here
  add-account : Bank, Customer → Bank
  update-accts : Acct-Class, Acct → Acct-Class
  update-cust-acct : Cust-Acct, CA-Link → Cust-Acct
events
  new-bank : → Bank
  start-account : Bank, Customer, Acct-No → Bank
  make-deposit, make-withdrawal : Bank, Acct-No, Amnt → Bank
axioms
  definition of aggregate methods in terms of components here
  (...see text for example ...)
 % invariants
  ∀ (a: Acct-Class, c: Cust-Class) size(a) ≥ 0 ∧ size(c) ≥ 0;
  ∀ (c: Cust-Acct-Class, b: Bank, a: Acct-No) size(c) ≥ 0 ∧ balance(b,a) ≥ 0;
 % definition of operations
  ∀ (b,b1: Bank) attr-equal(b,b1) ⇔ acct-obj(b) = acct-obj(b1)
    ∧ cust-obj(b) = cust-obj(b1) ∧ cust-acct-assoc(b) = cust-acct-assoc(b1);
  ∀ (b: Bank, a: Address, an: Acct-No, c: Customer) size(image(acct-obj(b),c,an)) = 1
    ⇒ singleton(a) = image(acct-obj(b),c,an) ∧ balance(b,an) = bal(a);
 % definition of methods
  acct-obj(create-bank()) = create-acct-class();
  cust-obj(create-bank()) = create-cust-class();
  cust-acct-assoc(create-bank()) = create-cust-acct();
  ∀ (b, b1: Bank, an: Acct-No, c: Customer)
    add-account(b, c, an) = b1
      ∧ acct = new-acct(date) %% date built in
      ∧ acct-obj(b1) = update-accts(acct-obj(b), acct)
      ∧ cust-acct = new-cust-acct(cust, acct, acct-no)
      ∧ cust-acct(b1) = update-cust-acct(cust-acct(b), cust-acct);
 % definition of events
  attr-equal(new-bank(), create-bank());
  ∀ (b:Bank, an:Acct-No, c:Customer) attr-equal(start-account(b,c,an),add-account(b,c,an));
  ∀ (b: Bank, an: Acct-No, c: Customer, am: Amount) size(image(acct-obj(b),c,an)) = 1
    ⇒ attr-equal(image(cust-acct-assoc(make-deposit(b,an,am),c,an)),
      deposit(image(cust-acct-assoc(b),c,an),am));
  ∀ (b: Bank, an: Acct-No, c: Customer, am: Amount)
    size(image(acct-obj(b),c,an)) = 1 ∧ balance(b,an) ≥ x
    ⇒ attr-equal(image(cust-acct-assoc(make-withdrawal(b,an,am))),
      withdrawal(image(cust-acct-assoc(b),c,an),am))
end-class
```

Fig. 16. Aggregate specification.

respond to external events. Then, constraints between aggregate components are specified to ensure that the aggregate does not behave in an unsuitable or unexpected manner. Finally, local event communication paths are defined. The definition of new functions and constraints is discussed in this section, while communication between objects is discussed in Section 7.

6.2.1 Specification of Functionality

In an aggregate, components work together to provide the desired functionality. Functional decomposition, often depicted using data flow diagrams (DFDs), is used to break aggregate-level methods into lower-level processes. Processes defined in the functional model are mapped to

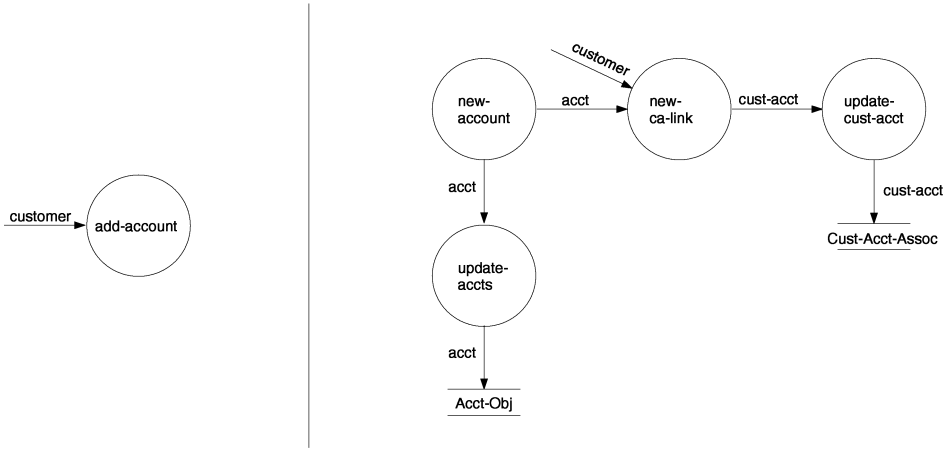


Fig. 17. Bank aggregate functional decomposition.

events and attributes defined in the aggregate components through aggregate-level axioms.

An example is shown by the data flow diagram in Fig. 17 for the aggregate method *add-account*, used to implement aggregate event *start-account*. The *make-deposit* and *make-withdrawal* events map directly to component events and do not require further functional decomposition.

The *add-account* process adds an account for an established customer and is defined in terms of operations defined directly in the bank specification (Fig. 16) or included into the bank specification via the bank aggregate specification from the customer-account link specification (Fig. 13) and the account specification (Fig. 4). The following axiom defines *add-account* in terms of these subprocesses and data flows as depicted in the data flow diagram.

$$
\begin{aligned}
&\forall(b, b1 : Bank, c : Customer)\\
&\exists(acct : Acct, cust\text{-}acct : CA\text{-}Link)\\
&add\text{-}account(b, c) = b1\\
&\quad \wedge\, acct = new\text{-}acct(date)\ \%\% \text{ assume date is built in}\\
&\quad \wedge\, acct\text{-}obj(b1) = update\text{-}accts(acct\text{-}obj(b), acct)\\
&\quad \wedge\, cust\text{-}acct = new\text{-}ca\text{-}link(c, acct)\\
&\quad \wedge\, cust\text{-}acct(b1) = update\text{-}cust\text{-}acct(cust\text{-}acct(b),\\
&\qquad\qquad cust\text{-}acct);
\end{aligned}
$$

The *add-account* method has two parameters, the bank object, b, plus an existing customer object, as shown in the data flow diagram, and returns the modified bank object, $b1$. The *add-account* method is defined by its subprocesses. First, a new account *acct* is created by invoking the *new-acct* process. This is passed to the *update-accts* process, which stores the new account in the account class, and to the *new-ca-link* process, along with the customer, which returns a cust-acct link. Finally, the new cust-acct link is passed to the *update-cust-acct* process, which stores it in the cust-acct association. The *new-acct* and *new-ca-link* processes are the events defined in the Acct class and the CA-Link association, respectively, and are already available via the aggregate. The *update-accts* and *update-cust-acct* processes could already exist as part of the account class and cust-acct association, but, as shown here, are defined in the aggregate specification.

6.2.2 Specification of Constraints between Components

In an aggregate, component behavior must often be constrained if the aggregate is to act in an integrated fashion. Generally, these *constraints* are expressed by axioms defined over component attributes. Because the aggregate is the colimit of its components, the aggregate may access components directly and define axioms relating various component attributes.

A simplified automobile object diagram is shown in Fig. 18. The object diagram contains one engine with an *RPMs* attribute, one transmission with a *Conversion-Factor* attribute, and four wheels, each with an *RPMs* attribute. Two relationships exist between these objects, *Drives*, that relates the transmission to exactly two wheels, and *Connected*, that relates two wheels (probably by an axle). Obviously, there are a number of constraints implicit in the object diagram that must be made explicit in the aggregate. First, the *RPMs* of the engine, *Conversion-Factor* of the transmission, and *RPMs* of the wheels are all related. Also, the wheels driven by the transmission must be "connected," and all "connected" wheels should have the same RPMs. The axiom

$$
\begin{aligned}
&\forall(a : Automobile, e : Engine, t : Transmission, d : Drives)\\
&\qquad e \in engine\text{-}obj(a) \wedge t \in transmission\text{-}obj(a)\\
&\qquad\qquad \wedge\, d \in drives\text{-}assoc(a)\\
&\quad \Rightarrow rpm(wheel\text{-}obj(d)) = rpm(e) * conversion\text{-}factor(t)
\end{aligned}
$$

defines the relationship between the *RPMs* of the wheels driven by the transmission, the transmission *conversion-factor* and the engine *RPMs*. In this case, *wheel-obj* is an object valued attribute of a *drives* link that points to the two wheels connected to the transmission. The axiom

$$
\begin{aligned}
&\forall(c : Connected) c \in connected\text{-}assoc(a) \Rightarrow rpms(wheel1(c))\\
&\qquad = rpms(wheel2(c));
\end{aligned}
$$

ensures that the two wheels connected by a *connected* link have the same *RPMs* values (here, *wheel1* and *wheel2* are the object valued attributes of the link). The final constraint, that the two wheels driven by the transmission be

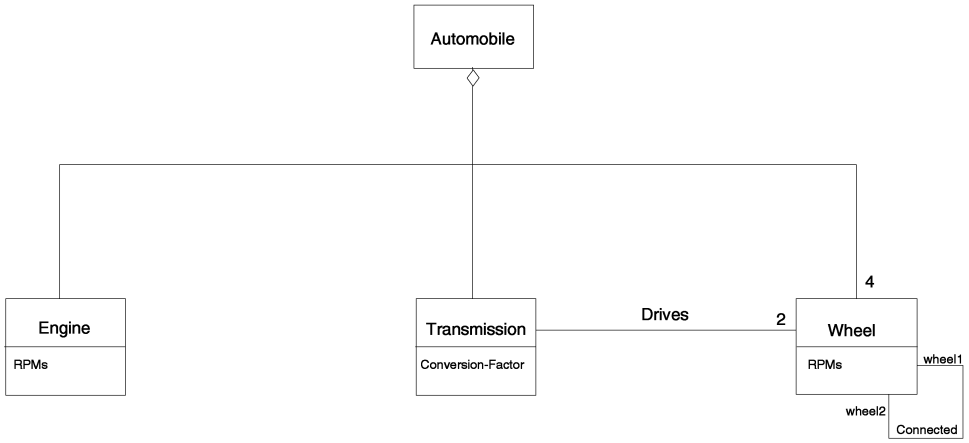


Fig. 18. Automobile aggregate functional decomposition.

connected, is specified implicitly in the specification of the *create-automobile* method. After the transmission and wheel objects ($w1$, $w2$, $w3$, and $w4$) are created in lines 1 through 5, *drives* and *connected* links are created and defined to ensure the appropriate constraints are met in lines 6 through 9. Finally, in line 10, the engine is created and inserted into the automobile aggregate.

$$
\begin{aligned}
& t = \textit{new-transmission}() && (1)\\
& \wedge\ \textit{transmission-obj}(\textit{create-automobile}()) \\
& \quad = \textit{insert}(t, \textit{new-transmission-class}()) && (2)\\
& \wedge\ w1 = \textit{new-wheel}() \wedge w2 \\
& \quad = \textit{new-wheel}() \wedge w3 = \textit{new-wheel}() \\
& \quad \wedge\ w4 = \textit{new-wheel}() && (3)\\
& \wedge\ \textit{wheels-obj}(\textit{create-automobile}()) = && (4)\\
& \quad \textit{insert}(w1, \textit{insert}(w2, \textit{insert}(w3, \\
& \quad \quad \textit{insert}(w4, \textit{new-wheels-class}())))) && (5)\\
& \wedge\ \textit{drives-assoc}(\textit{create-automobile}()) = && (6)\\
& \quad \textit{insert}(\textit{new- drives-link}(t, w2), \\
& \quad \quad \textit{insert}(\textit{new-drives-link}(t, w1), \textit{new-drives}())) && (7)\\
& \wedge\ \textit{connected-assoc}(\textit{create-automobile}()) = && (8)\\
& \quad \textit{insert}(\textit{new-connected-link}(w1, w2), \\
& \quad \quad \textit{insert}(\textit{new-connected-link}(w3, w4), \\
& \quad \quad \textit{new-connected}())) && (9)\\
& \wedge\ \textit{engine-obj}(\textit{create-automobile}()) = \\
& \quad \textit{insert}(\textit{new-engine}(), \textit{new-engine-class}()); && (10)
\end{aligned}
$$

Because wheels $w1$ and $w2$ are associated with the transmission via the *drives* association in the line 6, they are also associated together via the *connected* association in line 8. Thus, the constraint is satisfied whenever an automobile aggregate object is created.

7 Object Communication

At this point, our theory-based object model is sufficient for describing classes, their relationships, and their composition into aggregate classes; however, how objects communicate has not yet been addressed. For example, suppose the banking system described earlier has an *ARCHIVE* object which logs each transaction as it occurs. Obviously, the *ARCHIVE* object must be told when a transaction takes place. In our model, each object is aware of only a certain set of *events* that it generates or receives. From an object's perspective, these events are generated and broadcast to the entire system and received from the system. In this scheme, each event is defined in a separate event theory, as shown in Fig. 19.

An *event theory* consists of a class sort, parameter sorts, and an event signature that are mapped via morphisms to sorts and events in the generating and receiving classes. If an event is being sent to a single object, then the event theory class sort is mapped to the class sort of that object class. However, if the event theory class sort is mapped to the class sort of a *class set*, then communication may occur with a set of objects of that class. The other sorts in an event theory class are the sorts of event parameters. The final part of an event theory, the event signature, is mapped to a compatible event signature in the receiving class. The colimit of the classes, the event theory, and the morphisms unify the event and sorts such that invocation of the event in the generating class corresponds an invocation of the actual event in the receiving class.

To incorporate an event into the original *ACCT* class, the *ARCHIVE-WITHDRAWAL* event theory specification is imported into the *ACCT* class and an object-valued attribute, *archive-obj*, is added to reference the archival object. The axioms defining the effect of the withdrawal event are modified to reflect the communication with the *ARCHIVE* object as shown below.

$$
\begin{aligned}
& \forall(a : \textit{Acct}, x : \textit{Amnt})\textit{acct-state}(a) \\
& \quad = \textit{ok} \wedge \textit{bal}(a) \geq x \Rightarrow \textit{acct-state}(\textit{withdrawal}(a, x)) = \textit{ok} \\
& \quad \wedge\ \ \textit{archive-obj}(\textit{withdrawal}(a, x)) \\
& \quad \quad = \textit{archive-withdrawal}(\textit{archive-obj}(a), a, x) \\
& \quad \wedge\ \textit{attr-equal}(\textit{withdrawal}(a, x), \textit{debit}(a, x)); \\
& \forall(a : \textit{Acct}, x : \textit{Amnt})\textit{acct-state}(a) \\
& \quad = \textit{ok} \wedge \textit{bal}(a) < x \Rightarrow \textit{acct-state}(\textit{withdrawal}(a, x)) \\
& \quad \quad = \textit{overdrawn} \\
& \quad \wedge\ \textit{archive-obj}(\textit{withdrawal}(a, x)) \\
& \quad \quad = \textit{archive-withdrawal}(\textit{archive-obj}(a), a, x) \\
& \quad \wedge\ \textit{attr-equal}(\textit{withdrawal}(a, x), \textit{debit}(a, x));
\end{aligned}
$$

Basically, the axioms state that, when a *withdrawal* event is received, the value of the *archive-obj* is modified by the

event ARCHIVE-WITHDRAWAL is
class sort Archive
sorts Acct, Amnt
events
archive-withdrawal : Archive, Acct, Amnt → Archive
end-class

Fig. 19. Event theory.

archive-withdrawal event defined in the event theory specification. Thus, the *ACCT* object knows it communicates with some other object or objects; however, it does not know who they are. With whom an object communicates (or, for that matter, if the object communicates at all) is determined at the aggregate-level where the actual connections between communicating components are made.

The modified *BANK* aggregate diagram that includes the *ARCHIVE-WITHDRAWAL* event theory and an *ARCHIVE-CLASS* specification is shown in Fig. 20. The colimit operation includes morphisms from *ARCHIVE-WITHDRAWAL* to *ACCT-CLASS* and *ARCHIVE-CLASS* that unify the sorts and event signature in *ACCT-CLASS* with the sorts and event signature of *ARCHIVE-CLASS*. This unification creates the communication path between account objects and archive objects.

Communicating with objects from multiple classes requires the addition of another level of specification which "broadcasts" the communication event to all interested object classes. The class sort of a *broadcast theory* is called a broadcast sort and represents the object with which the sending object communicates. The broadcast theory then defines an object-valued attribute for each receiving class. Fig. 21 shows an example of the *ARCHIVE-WITHDRAWAL-MULT* event theory modified to communicate with two classes. In this case, the *ARCHIVE-WITHDRAWAL* theory is used to unify the *ARCHIVE-WITHDRAWAL-MULT* with the *ACCOUNT* class as well as the other two classes. A simplified version of the colimit diagram specification is shown in Fig. 22.

Multiple receiver classes add a layer of specification; however, multiple sending classes are handled very simply. The only additional construct required is a morphism from each sending class to the event theory mapping the appropriate object-valued attribute in the sending class to the class sort of the event theory and the event signature in the sending class to the event signature in the event theory.

7.1 Communication between Aggregate and Components

Communication between components is handled at the aggregate level as described above. However, when the communication is between the aggregate and one of its components, the unification of object-valued attributes and class sorts via event theories does not work since the class sort of the aggregate is not created until after the colimit is computed. The solution requires the use of a *sort axiom* that equivalences two sorts, as shown below:

$$sort\text{-}axiom\ sort1 = sort2.$$

Using the bank example discussed above, assume the *archive-withdrawal* event is also received by the Bank aggregate. The *archive-withdrawal* event theory is included in the Account class type and, by the colimit operation, the Bank aggregate. To enable the Bank aggregate to receive the archive-withdrawal event, a sort-axiom is used in the Bank specification to equivalence the *Bank* sort of the aggregate with the *Archive* sort from the event theory, as shown below:

$$sort\text{-}axiom\ Bank = Archive.$$

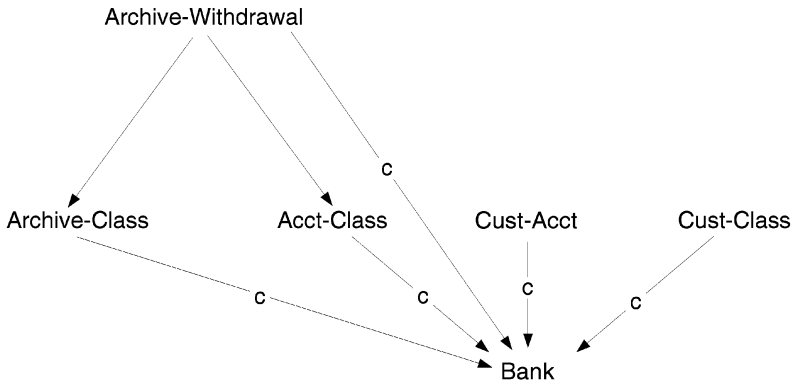


Fig. 20. Bank aggregate with archive.

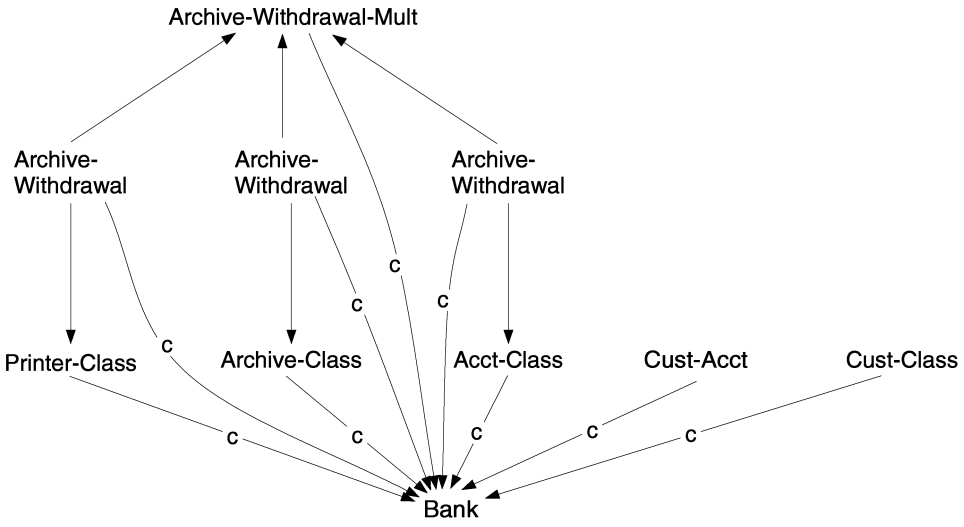


Fig. 22. Aggregate using a broadcast theory.

Use of the sort axiom unifies the *Bank* sort and the *Archive* sort and, thus, the signatures of the *archive-withdrawal* events defined in the event theory and the Bank aggregate become equivalent.

Communication from the aggregate to the components, or subcomponents, is much simpler. Since the aggregate includes all the sorts, functions, and axioms of all of its components and subcomponents via the colimit operations, the aggregate can directly reference those components by the object-valued attributes declared either in itself or its components. Because an aggregate is aware of its configuration, determining the correct object-valued attribute to use is not a problem.

8 Discussion of Results and Future Efforts

8.1 Object Model

Our research establishes a formal mathematical representation for the object-oriented paradigm within a category theory setting. In our theory-based object model, classes are defined as theory presentations or specifications and the basic object-oriented concepts of inheritance, aggregation, association, and interobject communication are formally defined using category theory operations. While some work formalizing aspects of object-orientation exists [8], [18], [23], [24], [25], ours is the first to formalize all the important aspects of object-orientation in a cohesive, computationally tractable framework. In fact, our formalization of inheritance, aggregation, and association, provides techniques for ensuring the consistency of

event ARCHIVE-WITHDRAWAL-MULT is
class sort Archive
sorts Amnt, Acct, X, Y
attribute
  x-obj : Archive → X
  y-obj : Archive → Y
events
  archive-withdrawal : Archive , Acct, Amnt → Archive
  archive-withdrawal : X, Acct, Amnt → X
  archive-withdrawal : Y, Acct, Amnt → Y
axioms
  ∀ (a: Archive, ac: Acct, am: Amnt)
    x-obj(archive-withdrawal(a,ac,am)) = archive-withdrawal(x-obj(a),ac,am)
    ∧ y-obj(archive-withdrawal(a,ac,am)) = archive-withdrawal(y-obj(a),ac,am)
end-class

Fig. 21. Broadcast theory.

object-oriented specifications based on the composition process itself.

The completeness of our integrated model allows the capture of any object-oriented model as a formal specification. Furthermore, the algebraic language O-SLANG allows for straightforward translation into existing algebraic languages, such as Slang or Larch, for further transformation into executable code. Thus, this model provides a bridge from existing informal CASE tools to existing formal specification languages, tying the ease of use of the former to the technical advantages of the latter.

8.2 Application of Object Model

To show the applicability of the theory-based object model, we developed a proof of concept parallel-refinement specification acquisition system. This system used a commercially available, OMT-based, object-oriented CASE tool to capture the informal specification. This included graphical representation of the object, dynamic, and functional models along with textual input in the form of method definitions and class-level constraints (neither of these have a graphical format defined in OMT and are generally easier to define directly using first-order axioms). The output from the user-interface was then parsed and translated into O-SLANG based on the theory-based object model. The translation from graphically-based input to O-SLANG was completely automated.

Two complete object-oriented domain models were developed using this system: a school records database and a fuel pumping station. These domains were chosen to demonstrate the wide diversity of domains, stressing both functional and dynamic aspects, supported by this model. In total, over 37 classes, including 76 methods and operations, 89 attributes, five aggregates, 47 events, and seven associations were specified. These domain models were sufficiently large and diverse to demonstrate the application of the theory-based model to support realistic problem domains.

8.3 Future Plans

The definition of theory-based models that can be mapped 1:1 to an informal representation provides the necessary framework for a parallel refinement system for specification development, as shown in Fig. 1. Our theory-based object model allows for the development of a domain model as a library of class theories. This O-SLANG representation can next be transformed into a Slang specification in a straightforward manner to allow the full use of the Specware development system. The Specware system has already demonstrated the ability to generate executable code from algebraic specifications. Thus, the technology now exists to transform informal object-oriented models to correct executable code.

While the class theories can be translated into code, the desired approach is to treat them as a full domain model. From this, a specific specification can be developed for input to design processing. Thus, the next step is the development of the *specification generation/refinement subsystem* in Fig. 1, an "elicitor-harvester" that will elicit requirements from a user by reasoning over the domain model and harvesting components of the domain model to build the desired specification. The ability to map between an informal model and the theory-based object model will allow the user to interface with the system using a familiar informal representation, while the formal model can support the reasoning needed to guide the user, as well as assuring that the harvested specification remains consistent with the constraints of the domain model.

ACKNOWLEDGMENTS

This work has been supported by grants from Rome Laboratory, the US National Security Agency, and the US Air Force Office of Scientific Research.

REFERENCES

[1] C. Green et al., "Report on a Knowledge-Based Software Assistant," *Readings in Artificial Intelligence and Software Eng.*, C. Rich and R. Waters, eds., pp. 377–428, San Mateo, CA: Morgan Kauffman, 1986.

[2] M.R. Lowry, "Software Engineering in the Twenty-First Century," *AI Magazine,* Fall 1992.

[3] D.R. Smith, "KIDS – A Semi-automatic Program Development System," *IEEE Trans. Software Eng.,* vol. 16, no. 9, pp. 1,024–1,043 Sep. 1990.

[4] D.R. Smith, "Transformational Approach to Transportation Scheduling," *Proc. Eighth Knowledge-Based Software Eng. Conf.,* pp. 60–68, Oct. 1993.

[5] J. Rumbaugh et al., *Object-Oriented Modeling and Design.* Englewood Cliffs, New Jersey: Prentice Hall, 1991

[6] R. Jullig and Y.V. Srinivas, "Diagrams for Software Synthesis," *Proc. Eighth Knowledge-Based Software Eng. Conf.,* 1993.

[7] M.D. Fraser, K. Kumar, and V.K. Vaishnavi, "Strategies for Incorporating Formal Specifications," *Comm. ACM,* vol. 37, pp. 74–86, Oct. 1994.

[8] R.H. Bourdeau and B.H. Cheng, "A Formal Semantics for Object Model Diagrams," *IEEE Trans. Software Engineering,* vol. 21, no. 10, pp. 799–821, Oct. 1995.

[9] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," *Science of Computer Programming,* vol. 8, pp. 231–274, 1987.

[10] T.C. Hartrum and P.D. Bailor, "Teaching Formal Extensions of Informal-Based Object-Oriented Analysis Methodologies," *Proc. Software Eng. Education,* Pittsburgh, Pa.: Software Eng. Education, Software Eng. Inst. (SEI), Jan. 1994.

[11] S.A. DeLoach, "Formal Transformations from Graphically-Based Object-Oriented Representations to Theory-Based Specification," PhD thesis, Wright-Patterson AFB: US Air Force, Inst. of Technology, AFIT/DS/ENG/96-05, AD-A310 608, June 1996.

[12] Kestrel Institute, *Slang Language Manual: Specware Version Core4,* Oct. 1994.

[13] J.A. Goguen and R.M. Burstall, "Some Fundamental Algebraic Tools for the Semantics of Computation Part I: Comma Categories, Colimits, Signatures and Theories," *Theoretical Computer Science,* vol. 31, pp. 175–209, 1984.

[14] K. Lano and H. Houghton, "Specifying a Concept-Recognition System in Z++," *Object-Oriented Specification Case Studies,* K. Lano and H. Houghton, eds., pp. 137–157, Prentice-Hall, 1994.

[15] D. Carrington et al., "Object-Z: An Object-Oriented Extension to Z," *Proc. Formal Description Techniques, II: Proc. IFIP Second Int'l Conf. Formal Description Techniques for Distributed Systems and Comm. Protocol,* pp. 281–297, Dec. 1989.

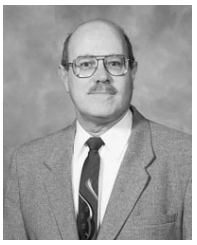
[16] K. Lano and H. Houghton, "A Comparative Description of Object-Oriented Specification Languages," *Object-Oriented Specification Case Studies,* K. Lano and H. Houghton, eds., pp. 20–54, Prentice Hall, 1994.

[17] A.J. Alencar and J.A. Gougen, "Specification in OOZE with Examples," *Object-Oriented Specification Case Studies,* K. Lano and H. Houghton, eds., pp. 158–183, Prentice Hall, 1994.

[18] J.A. Goguen and J. Meseguer, "Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics," *Research Directions in Object-Oriented Programming,* B. Shriver and P. Wegner, eds., pp. 417–477, MIT Press, 1987.

[19] J.A. Goguen and T. Winkler, "Introducing OBJ3," Techical Report , Computer Science Laboratory, SRI Int'l, Menlo Park, Calif., Aug. 1988.
[20] Y.V. Srinivas, "Algebraic Specification: Syntax, Semantics, Structure," Techical Report TR 90-15, Dept. of Information and Computer Science, Univ. of California at Irvine, June 1990.
[21] Y.V. Srinivas, "Category Theory Definitions and Examples," Techical Report TR 90-14, Dept. of Information and Computer Science, Univ. of California at Irvine, Feb. 1990.
[22] B. Liskov, "Data Abstraction and Hierarchy," *Proc. Conf. Object Oriented Programming Systems Languages and Applications (OOPSLA),* 1987.
[23] T. Bar-David, "Practical Consequences of Formal Defintions of Inheritance," *J. Object-Oriented Programming,* vol. 5, pp. 43–49, July/Aug. 1992.
[24] X.-M. Lu and T.S. Dillon, "An Algebraic Theory of Object-Oriented Systems," *IEEE Trans. Knowedge and Data Eng.,* vol. 6, no. 3, pp. 412–419, June 1994.
[25] H. Lin and M. chi Pong, "Modelling Multiple Inheritance with Colimits," *Formal Aspects of Computing,* vol. 2, pp. 301–311, 1990.

Scott A. DeLoach received his BS in computer engineering from Iowa State University in 1982, and his MS and PhD in computer engineering from the US Air Force Institute of Technology in 1987 and 1996. He is currently an assistant professor of computer science and engineering at the US Air Force Institute of Technology (AFIT). His research interests include design and synthesis of multiagent systems, knowledge-based software engineering, and formal specification acquisition. Prior to coming to AFIT, he was the technical director of Information Fusion Technology at the US Air Force Research Laboratory from 1996 to 1998. From 1987 to 1993, he was chief of Systems Engineering and Electronic Combat Support at Headquarters, Strategic Air Command and was a computer resources engineer at Wright-Patterson AFB, Ohio, from 1982 to 1986. He is a member of the IEEE Computer Society.



Thomas C. Hartrum received the BSEE and MS degrees in 1969, and his PhD in 1973 from Ohio State University, and an MBA in 1979 from Wright State University. He is currently an associate professor in the Department of Electrical and Computer Engineering at the US Air Force Institute of Technology (AFIT), where he has been a faculty member for 20 years. He is an adjunct associate professor at Wright State University and teaches software engineering there part time. He has been involved in software engineering for 11 years and specifically with knowledge-based software engineering (KBSE) since 1990. Prior to AFIT, he was a research engineer at the US Air Force, Aerospace Medical Research Laboratory. He is a member of the IEEE and the IEEE Computer Society