Modeling Behavior, a Step Towards Defining Functionally Correct Views of Complex Objects in Concurrent Engineering

Fawaz S. Al-Anzi David L. Spooner
Computer Science Department
Rensselaer Polytechnic Institute
Troy, New York, 12180

Abstract

Multidisciplinary concurrent engineering needs to model and manage different views of complex designs. Previous attempts to address the problem of creating views of complex objects in object oriented database systems focus on the structure of complex objects; little attention is paid to how complex object behavior is effected when creating views. We believe that designing functionally correct behavior for a complex object should be a major consideration when defining a view to guarantee correctness of the derived classes.

In this paper, we study the problem for designing functionally correct views of complex objects in concurrent engineering. View behavioral modeling requirements are presented. A behavior model that satisfies these requirements is presented. This model is demonstrated on an example complex object that represents process management.

1 Introduction

The concurrent engineering methodology is a recent technological development aimed at reducing the length of the traditional serial design process, hence, reducing the the expense of a product, by allowing multiple engineers from different disciplines to work concurrently on an engineering design. Each engineering discipline participating in the design of a product has its own perspective on the design. This implies that each engineering discipline needs to optimize the product design to a form that best serves its perspective.

Object oriented database technology has shown its promise in supporting multidisciplinary concurrent engineering needs to model and manage complex designs through its modeling power and natural integration with the object oriented programming paradigm[20]. Capturing of the semantics of a design through associations, data protection by encapsulation and the convenience of reuse via inheritance make it even more suitable to be used as the basis for multidiscipline concurrent engineering technology in which each design can be modeled as a complex object.

However, object oriented database technology has revealed shortcomings when used to restructure design data to support design perspectives, and it includes limited mechanisms to effectively integrate these perspectives. A natural way to overcome these shortcomings is to introduce a view mechanism for complex objects that provides the required design perspectives.

Conventional relational database views are defined by declarative queries which are evaluated as needed. This is a simple task because views modeled in relational databases are simple tables and languages in relational databases use declarative queries. On the other hand, language constructs in object oriented systems tend to be procedural rather than declarative. For example, the EXPRESS language[21], which is becoming an international standard for modeling engineering data, has constructs such as procedures, functions, loops, assignments, *etc.* which are found in imperative languages. Any language that is less powerful will be a handicap to the manipulation of the complex network of objects needed to model engineering data. Hence, declarative views most likely will not be enough to create views of complex objects for concurrent engineering. Rather, views of complex object will be constructed using a more powerful procedural approach. To aid the user in creating such views, sophisticated tools will be responsible for automating most parts of the view definition process. These tools will aid in the construction of a view, the evolution of a view, and keeping a view consistent with the underlying database as the underlying database evolves. Putting all this together, a Design System for Views of Complex Objects (DSVCO) is needed.

The DSVCO we envision has three tasks; customization of virtual classes to allow the user to define his/her own classes from existing classes, integration of virtual classes into one consistent global schema with the original classes, and the specification of functionally correct views of complex objects on the global schema. Hence, the architecture of a design system for views of complex objects can be defined as shown in Figure 1. It takes a base schema (Original Schema in the figure) as input and produces the proper view schemas for different application programs. The Extended Schema Constructor and the Schema Evolution Constructor create an intermediate schema (Extended Schema) that integrates the base schema and virtual classes defined by the user. This process is called extended schema evolution. The View Constructor and the Extended Schema Evolution Monitor then create the different views that are interfaced to the application programs.

Early attempts to address views in object oriented databases in the research literature focus on designing views

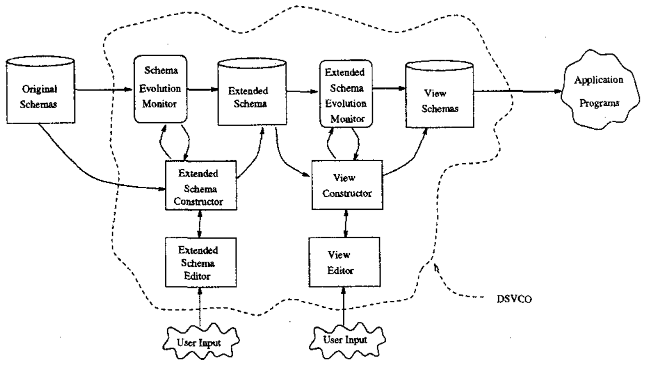

Figure 1: Architecture of a Design System for Views of Complex Objects (DSVCO).

of single objects. Of these attempts we mention the following.

The FUGUE object model proposed by Heiler and Zdonik investigates view management as a combination of the object oriented model and the functional data model [8]. The FUGUE system defines views as a set of abstract types which are derived from the base types. The operations performed on the view objects are transformed to operations on the base types.

Multiple interfaces to class objects is proposed by Shilling and Sweeney [18]. Their approach limits the access rights to functions and attributes through these interfaces. Their work focuses on individual classes since their work is programming oriented.

Schiefer[16] defined customized interfaces for an object-oriented database. His approach requires the explicit declaration of inheritance relationships between view classes. His work is done in the context of the STONE project (STructured and Open Environment).

Abiteboul and Bonner[1] propose using a functional approach for defining views. They try to overcome the problem of a model lacking flexibility by proposing a sophisticated view mechanism that is capable of creating imaginary objects, virtual classes, and virtual attributes; however, not enough exploration of functions is done. Although their work contributes to defining OO views, their work is mostly informal.

Recent work has started to realize that the modeling power of the object oriented paradigm is not only a result of inheritance and data encapsulation, but also a result of the capturing of the semantics of complex objects through the use of associations between the subparts of a complex object. Researchers have started to investigate how semantic information can be manipulated by changing associations between objects in a view. Of these attempts we mention the following.

Tusda, Yamamoto, Hirakawa, Tanaka, and Tadao [22] propose to increase flexibility by introducing structural messages which are used to change the structure of objects in the schema. They also define the concept of classtype, which is used to define object properties without changing the class hierarchy. Their work does not provide a formal view mechanism, rather, they try to solve the problems that arise when restructuring the schema.

The MULTIVIEW project proposed by Rundensteiner [13] is a recent attempt to address views in object oriented databases by formalizing the concept of orthogonality between sets and types. She defines an object algebra for manipulating a class hierarchy. Although her model clearly differentiates between the set concept and the type concept for a class, her object algebra operators do not separate the two concepts. Also, her work emphasizes the restructuring of *IS-A* associations in the object hierarchy without considering how to restructure other types of associations in the hierarchy.

Liu's[11] work is another recent attempt to address views in object oriented databases. He defines two different manipulating algebras. The first is to deal with *IS-A* associations and the second is to deal with other associations between classes. This work constitutes the most complete set of algebras to restructure an object hierarchy for both *IS-A* and other types of associations. However, it does not include a methodology on how to use the algebras to design the views.

In general, the previous attempts propose models that are designed to capture the structure of views. Little attention is paid to how object methods, simple object constraints, and complex object constraints (rules) are effected when creating views of complex objects. Designing functionally correct behavior (methods and constraints) should be a major consideration when defining a view to guarantee correctness of the behavior for derived classes in the view.

In order to achieve a good understanding of behavior in views of complex objects, the following questions need to be addressed.

- What behavior is needed in views of complex objects and how is it defined?
- How much of it can be derived from the original database?
- How do we validate that the behavior of a view is functionally correct?

In this paper we study these questions.

1.1 Object Model

The object model we use is built on the model described by Liu [11]. A class in our object model is defined by a 4-tuple $C = \{ATTR, ASSO, FUNC, RULE\}$, where $ATTR$ is a set of attribute and domain pairs. A domain is restricted to the primitive data types such as integer, real, *etc*, or to the types constructed from primitive data types by the type constructors such as array, list, *etc*. An attribute having another object class as its domain, which is traditionally used to model associations, is not allowed in our attribute definition. Associations are modeled by $ASSO$ as a set of 6-tuples: $\{AssoName$, $C1Name$, $C2Name$, $AssoType$, $Cardinality$, $Order\}$, with each 6-tuple in the set specifying that class $C1Name$ is associated with class $C2Name$ under the association named $AssoName$. $AssoType$ defines the association type. $Cardinality$ defines the multiplicity of the association. $Order$ specifies if the association is ordered or not. Separating associations from class attributes is done so that an algebra can manipulate associations independently of class attributes. $FUNC$ is a set of functions (methods) defined on C. $RULE$ specifies both structural constraints and behavioral constraints for instances of class C. An object o is identified by a unique object identifier (OID). Object o can be instantiated in multiple classes in a class hierarchy simultaneously (i.e., in a class and all its super classes). The instance of o in each class is identified by a unique instance identifier (IID) that is composed of the object's OID and a class identifier.

A complex object schema in an object oriented database is a directed and labeled multigraph $SG=(C,A)$, where C is a set of vertices representing object classes. A is a set of edges representing the associations between classes. A is partitioned into two sets. The first set is the *IS-A* relationships (associations). The subgraphs of SG connected by *IS-A* relationships form one or multiple class lattices. The second set includes all other types of associations. These edges are labeled by the association names they represent so that if more than one association exists between two classes, the associations can be identified by their names. Self loop associations are allowed in the latter set because a class could have an association with itself.

For example, Figure 2 shows a class schema that consists of ten classes. Some of these classes are associated via *IS-A* associations that form an *IS-A* lattice and define the subclass/superclass relationships, i.e., student is a person, undergraduate is a student, graduate is a student, employee is a person, professor is an employee, import is a car, domestic is a car, and Toyota is an import. The other associations represented by dashed lines are not *IS-A* associations, but instead model part of the semantics of the complex object, i.e., a person is a child of another person, some professor is the academic advisor of a student, some professor is the research advisor of a graduate student, and some people drive cars.

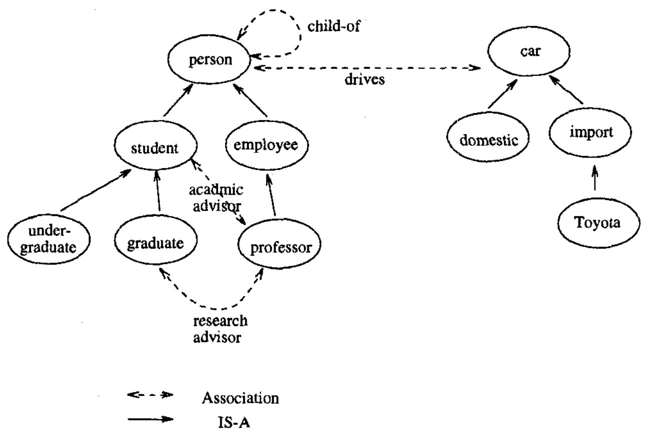

Figure 2: Class Schema example.

2 Behavior Model

Behavioral specification languages provide an understanding of the application domain, problem formulation, and solutions for a particular database. Also they play useful roles in design and the development cycle as well as serving as computer based tools for analyzing designs.

Languages for specifying system behavior can be categorized into two types. The first is languages based on abstract data types, e.g., Tecton [9], and Z [19]. The second is languages based on finite state machine for modeling reactive systems and concurrent engineering, e.g., Petri Nets [12], Statecharts [4], Life Cycle [15], and Behavior Diagrams [17][1].

Large scale specification of complex systems needs appropriate abstractions for system design, simple specification technology, and automated reasoning technology. We choose finite state machine models for our problem because they are more concrete in the sense that a reasonable amount of practical work is done in this field, more readable since a graphical notation is preferred over logical formulas, and more computation-oriented since executable specifications are preferred over non-executable abstract specifications.

[1]Some systems can be categorized into both types such as LOTUS [10].

We start by selecting the proper behavior model and extending it to model behavior of complex objects. The models we choose from are Petri nets, Statecharts, Objectcharts, Life Cycle, and Behavior Diagrams.

J.L. Peterson [12] introduced *Petri nets* as a graphical and mathematical tool particularly appropriate for systems which exhibit concurrent, asynchronous behavior. As a mathematical tool Petri nets allow the development of state equations and other mathematical models to study system behavior. Petri nets have been extended in many ways. Two extensions are of special interest to us. They are *Behavior Diagrams* presented by M. Schrefl [17], and the *Life Cycle* model presented by H. Sakai [15].

Statecharts, developed by D. Harel [7], overcome the problems of using finite-state machines to model reactive system behavior. D. Coleman, F. Hayes and S. Bear [4] propose extending the Statecharts model to characterize the behavior of an object class as a state machine. They call their model *Objectcharts*. It is a semantic model for introducing class behavior as the basis for subtyping.

2.1 Requirements

We survey the behavior models listed above according to the requirements for modeling the behavior of complex objects in concurrent engineering. The requirements are highly influnced by the way views of complex objects are defined. Views, in the behavioral sense, are refinements of the base behavior of the complex object. This refinement may include altering the way objects are grouped together, changing the operations that certain group of objects are subjected to, changing the level of detail that the trace of an object undergoes to change from one grouping to another, adding a new behavior, or hiding some of the base behavior. Using this prespective for defining views, the features that we will use to contrast the models are:

- *Function Representation*: Functions should be modeled as separate entities, where constraints can graphically be placed on them. A node representation for functions in the behavior graph is preferred over an arc representation so that constraints can be represented as arcs between functions.
- *Single Object States*: This allows the objects in a class to be partitioned into groups of objects according to their attribute values. In other words, a state is modeled as a predicate defined on the attributes in the object's class. Each group of objects has a set of operations that take an object from one group to another. This gives us a specific behavioral trace mechanism as objects undergo operational changes.
- *Inter-Object States*: This enables modeling valid combinations of states between two objects connected via an association. Constraints that relate values of attributes for objects in different classes in a complex object will be modeled using complex states. A complex object's state is modeled as a predicate defined over the attributes of the classes that make up the complex object.

- *Simple Functional Constraints*: To model the changes caused by functions and to keep the behavior graph of an object in a valid state, these constraints are used to define the pre-state and post-state requirements of every invocation of a function.
- *Association Modeling*: Because we are modeling complex objects as a collection of simple objects and associations between them, a correct modeling of associations is needed for complex object constraints in our model. Such modeling will provide the ability to capture changes to behavior graphs as associations between objects are altered, removed or added.
- *State Hierarchies*: These are needed to model the refinement process defined for behavior graphs as views with different levels of abstraction. These state hierarchies reduce the exponential growth of a model as design complexity increases.
- *Transition Hierarchy*: A functional hierarchy is desired so that the refinement process can rewrite a function as a network of substates and subfunctions between these substates. This hierarchy is used to refine transitions in the behavior graph for views at different levels of abstraction.
- *Complex Constraints*: Functions and states in different object classes should be able to constrain each other if classes are associated through an association. These inter-object constraints constitute the building blocks for defining the overall complex behavior.
- *AND/OR states*: The model should be able to define sections of the behavior graph that relate to other sections of the behavior graph in the sense that some parts are to be mutual exclusive, ored, anded, *etc.*
- *State/Transition Refinement*: It is necessary to support refinement from coarse states to substates to study behavior of restriction/abstraction view classes.
- *Complex Object Invariant*: We do not expect any of the models to support this feature; however, ideally a model should express the state of a complex object as valid or invalid.

Table 1 compares the five behavior models for each of these features.

In general, the surveyed models do not fully support our requirements for modeling the functional behavior of complex objects. The first missing concept is inter-object states. The second missing concept is behavior modeling of associations between classes. Third, complex constraints between objects in different classes are not expressed (except in the Life Cycle model). Fourth, complex object invariants that can be used to define valid complex object instances are not considered.

These four omissions require us to build our own model of complex object behavior. It is clear that the Life Cycle model is the closest to our target behavior model. In the following section we extend the Life Cycle model to be able to model complex object behavior.

2.2 Extended Life Cycle Model

We define behavior of a simple object in our model by its life cycle which expresses the process of creation, state transitions, and elimination of objects. This is based on the Life Cycle model[15] which is defined as follows. A set of values of all the attributes of an object at a certain point of time is called a *state*, and an operation that brings about changes of states is called a *transition*. Every object undergoes changes of states obeying a defined life schema pattern.

Let $B = (U, T, A)$ denote a bipartite graph, where U and T are disjoint sets of symbols called states and transitions, respectively, and A is a set of directed edges such that $A \subset (U \times T) \cup (T \times U)$. The following sets are defined for B and for an element u of $U \cup T$.

$$pre(u) = \{v | v \in U \cup T, (v, u) \in A\}$$

$$post(u) = \{v | v \in U \cup T, (u, v) \in A\}$$

$$sources(B) = \{u | u \in U \cup T, pre(u) = \phi\}$$

$$sinks(B) = \{u | u \in U \cup T, post(u) = \phi\}$$

A behavior graph[2] B in our model is a graph in which a set ACT is defined as

$$ACT = \bigcup_{\forall t \in T-(sources(B) \cup sinks(B))} ACT(t),$$

where $ACT(t) \subset pre(t) \times post(t)$ for a transition t. Every element of $ACT(t)$ is called an *activation* of t and constitutes a valid invocation of the transition t.

We write the behavior of a set of objects C that share the same properties (i.e., a class) as a behavior graph $B = (U, T, A, ACT)$ in which the conditions

$$sources(B) \cup sinks(B) \subset T$$

and

$$|sources(B)| = |sink(B)| = 1$$

hold. These conditions mean that no state is allowed to be a sink or a source in a behavior graph. Also, there is only one producing transition that creates class instances and one consuming transition that destroys classes instances per class.

A life cycle diagram is a *Petri* net graph representation of the life cycle schema. Figure 3 shows a simple schema example of an execution process in an operating system which is composed of three classes[3], *cpu*, *task*, and *I/O*, that are associated through associations *processed*, *use_IO*, and *coupled*. While this example is not directly related to concurrent engineering, it was chosen because it is familiar to most readers and simple enough to clearly illustrate the concepts. Figure 4 represents the behavior graph of the three classes. The circles represent states and the rectangles represent transitions. Arrows to node u in the graph come from all elements in the set $pre(u)$. Arrows also go from a node u to all elements in the set $post(u)$.

An object in a state in the $pre(t)$ set for a transition t undergoes a change to a new state in $post(t)$ if t is invoked by sending a message to the object. If there are closed paths in the graph, repeated states of the same object may occur. The transitions of *sources(B)* and *sinks(B)* represent the *producing* and *consuming* operations for objects.

The states of objects can be nested, overlapped, or disjoint. If a state can be further divided into substates and

[2] Sakai [15] defines the *behavior graph* as: The graph B in which the conditions $|pre(t)| = 1$ and $|post(t)| = 1$ hold, where $|x|$ denotes the cardinality of a set x. Our definition of behavior graph is more general since we allow a transition t to have more than one activation.

[3] In the diagram $\diamond$ is used to show that a part is an aggregation of other subparts.

Table 1: Summary of features of the five models.

Feature	Petri Nets [12]	Statecharts [7]	Objectcharts [4]	Life Cycle [15]	Behavior Diagrams [17]
Function Representation	node	arc	arc	node	node
Single Object States	yes	yes	yes	yes	yes
Inter-Object States	no	no	no	no	no
Simple Functional Constraints	yes	yes	yes	yes	yes
Association Modeling	no	no	no	no	no
States Hierarchy	no	yes	yes	yes	yes
Transition Hierarchy	no	yes	no	no	yes
Complex Constraints	no	no	no	yes	no
AND/OR States	no	yes	yes	yes	yes
State/Transition Refinement	no	yes	yes	yes	yes
Complex Object Invariant	no	no	no	no	no

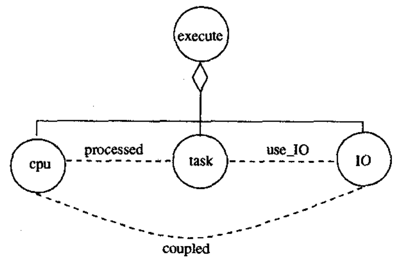

Figure 3: An abstract schema for execution of a task.

transitions, we say the state is *abstract*. If a state cannot be divided further, we say the state is *basic*. Similarly, we define an *abstract* transition as a transition that can be divided further into states and transitions. If a transition cannot be divided further, we say the activity is *basic* [17].

Every object has the abstract state named *exist* representing the fact that the object has been created. We artificially define a basic state named *not-exist* to express that the object is not yet produced as a member of the class, or has been consumed.

When an object enters a state u, values of certain attributes of the object may be updated. The constraints on attribute values at the time of a transition to a state u are called the ***attribute constraints*** associated with the state u. These attribute constraints constitute the basis for the *state description* of the state, which is defined in terms of boolean expressions of attribute constraints. On the other hand, a *transition description* is a set of *prestate* and *poststate* pairs for the transition. For example, in Figure 4, the state description of the state *cpu_waiting* in class *task* can be expressed by the boolean expression $(cpu\_status == cpu\_waiting \wedge IO\_status == IO\_not\_needed)$, where *cpu_status* and *IO_status* are attributes of the class *task*. Also in the same figure, the transition *get_cpu* has the set of two activations $\{(cpu\_waiting, using\_cpu), (start, using\_cpu)\}$ as its transition description. The following example is a specification of the simple object behavior for the classes in Figure 4.

2.3 Example

Consider the classes *task, I/O* and *cpu* in Figure 4. The class definitions, including the behavior for these objects, are described in Figures 5 and 6.

In this description, **Attributes** indicate the data store

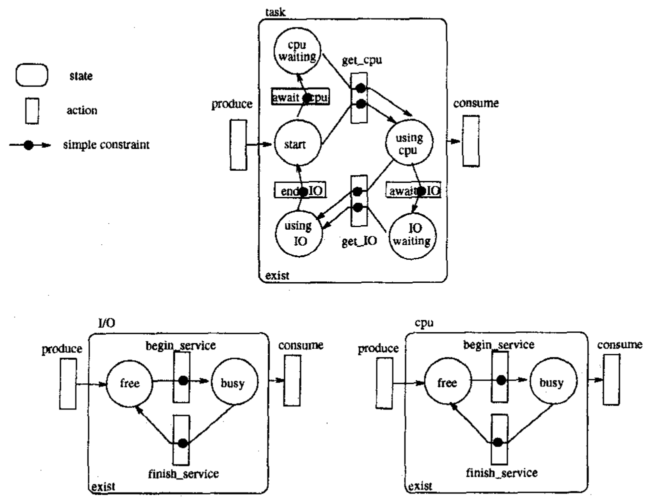

Figure 4: A basic life cycle of the three classes.

of the class, and **Transitions**, **States**, and **Activations** describe the behavior of the class instances as simple objects. The class *task* has two attributes, *cpu_status* and *IO_status*. Since states constrain domain attributes, different states of the class instances are described by the States part of the class. A *task* class instance can be in one (or more) of the following states; *cpu_waiting, served, IO_waiting,* and *using_IO*. The states are described by a predicate. If the predicate holds for some instance of the class, then the instance is in that state. An instance can satisfy multiple state descriptions. These corresponding states that an instance belongs to can be overlapping or nested but not disjoint.

Transitions (functions) to manipulate and update the attributes in the class *task* are *await_cpu, get_cpu, await_IO, get_IO,* and *end_IO*. The effect of each transition is value assignment to one or both attributes. There are two special transitions, *produce* and *consume,* used to create and destroy instances of the class.

Activations $Act_0 \ldots Act_8$ describe the behavior of a transition within an instance of the class *task* by stating pairs of prestates and poststates for every transition. These are the constraints on transitions, i.e., transitions can only be invoked on an instance that is in a state that belongs to a prestate in one of the pairs. The resulting state must be in the post state of the same pair. (If an object does not belong to the post state specified in the pair after the function

```
Class task;
{
        Attributes:
                cpu_status =cpu_not_needed,      domain {cpu_not_needed,cpu_waiting,enjoy_cpu}
                IO_status =IO_not_needed,      domain {IO_not_needed,IO_waiting,enjoy_IO}
        Transitions:
                produce();
                consume();
                await_cpu(){cpu_status=cpu_waiting,IO_status=IO_not_needed;},
                get_cpu(){cpu_status=enjoy_cpu;},
                await_IO(){IO_status=IO_waiting,cpu_status=cpu_not_needed;};
                get_IO(){IO_status=enjoy_IO;};
                end_IO(){IO_status=IO_not_needed,cpu_status=cpu_waiting;};
        States:
                start={cpu_status==cpu_not_needed ∧ IO_status==IO_not_needed}
                cpu_waiting={cpu_status==cpu_waiting ∧ IO_status==IO_not_needed}
                using_cpu={cpu_status==enjoy_cpu ∧ IO_status==IO_not_needed}
                IO_waiting={cpu_status==cpu_not_needed ∧ IO_status==IO_waiting}
                using_IO={cpu_status==cpu_not_needed ∧ IO_status==enjoy_IO}
        Activations:
                Act0={await_cpu,(start,cpu_waiting)}
                Act1={get_cpu,(cpu_waiting,using_cpu)}
                Act2={get_cpu,(start,using_cpu)}
                Act3={await_IO,(using_cpu,IO_waiting)}
                Act4={get_IO,(using_cpu,using_IO)}
                Act5={get_IO,(IO_waiting,using_IO)}
                Act6={end_IO,(using_IO,start)}
                Act7={produce,(not_exist,start)}
                Act8={consume,(exist,not_exist)}
}
```

Figure 5: *task* class.

```
Class cpu;
{
        Attributes:
                cpu_status =free;        domain {busy,free}
                cpu_speed =10^6,         domain real
        Transitions:
                produce(),
                consume();
                begin_service(){cpu_status=busy;};
                finish_service(){cpu_status=free;};
        States:
                free={cpu_status==free}
                busy={cpu_status==busy}
        Activations:
                Act0={begin_service,(free,busy)}
                Act1={finish_service,(busy,free)}
}
```

```
Class IO;
{
        Attributes:
                IO_status =free;         domain {busy,free}
                IO_speed =10^3;          domain real
        Transitions:
                produce(),
                consume();
                begin_service(){cpu_status=busy;};
                finish_service(){cpu_status=free;};
        States:
                free={cpu_status==free}
                busy={cpu_status==busy}
        Activations:
                Act0={begin_service,(free,busy)}
                Act1={finish_service,(busy,free)}
}
```

Figure 6: *cpu* and *IO* classes.

that implements the transition executes, the transition will be aborted and the change caused by the transition will not be committed.) For example, the function *get_cpu* can be invoked for an instance in the *cpu_waiting* or *start* states and causes the instance of class *task* to move to the *using_cpu* state.

This methodology of invoking transitions allows the user to define different implementations (with different names) to execute on different states of an object. This can be handy in defining optimized versions of the same function in the same class, but for different states of the object instance.

The descriptions of the *cpu* and *IO* classes are analogous.

2.4 Behavior of Associations

To model the behavior of associations, a state that represents an association is added to the behavior graph that represents the association. Constraints are added to the behavior model for methods that traverse associations. Also, constraints that ensure the existence of the two instances of the two classes to be associated are needed.

Most importantly, an association is used to hold states that incorporate values of attributes in the two class instances associated through this association. For example, we define a state for the coupled association that ensures compatibility of the *IO* instance that is coupled to a *cpu* instance. The state *compatible* is true if the speed of the *IO* instance is no less than 10^{-3} times the speed of the *cpu* instance to maximize the performance of executing tasks. Then, we write the state as

$$compatible = \{IO_1.speed \geq 10^{-3} * cpu_1.speed\},$$

where IO_1 and cpu_1 are the two states associated through the *coupled* association.

Associations are also used as a place to hold complex constraints between two classes associated through the association. An instance of an association holds instances of these constraints between the object classes associated through the association. An example of complex constraints in an association is given in the next section after a formal definition of complex constraints.

2.5 Behavior of Complex Objects

A life cycle schema defines constraints on the behavior of a single object by adding them to the prestate of transitions that cause state changes. For example, the transition *get_cpu* in the previous example needs a *cpu* instance in the *free* state in order to change the *task* instance from the *cpu_waiting* or *start* states to the state *using_cpu*.

In general, there are close relationships between the life cycle of a complex object and those of the component objects and their associations that it contains, since the behavior of a complex object is a composition of the behavior of its component objects and associations. We view complex constraints as *constraints motivated by associations*, where it is necessary to maintain consistent relationships between the life cycles of the component objects that compose a complex object. To establish this consistency of life cycles, the following BNF definition of the general complex constraint is defined[4]:

Complex_form ::= < *term_list* > requires < *term_list* >
< *term_list* > ::= < *term* > | < *term* > < *term_list* >
< *term* > ::= < *state* > | < *activation* >

where a state predicate is defined by $state(o : C, u)$ for an object o of the class C in the state u. An activation is defined by $[o : C, t]$ where $t = \{a, (s_1, s_2)\}$ is the invocation of the transition a on an object o of the class C in state s_1, and o will be in the state s_2 when transition a completes.

Note that a state and an activation on the opposite sides of a form should be in two classes associated through an association, i.e., if a complex constraint exists between two classes C_i and C_j then there must exist at least one association $ASSO_k$ such that C_i $ASSO_k$ C_j. This association $ASSO_k$ is the place used to store the complex constraint form instance.

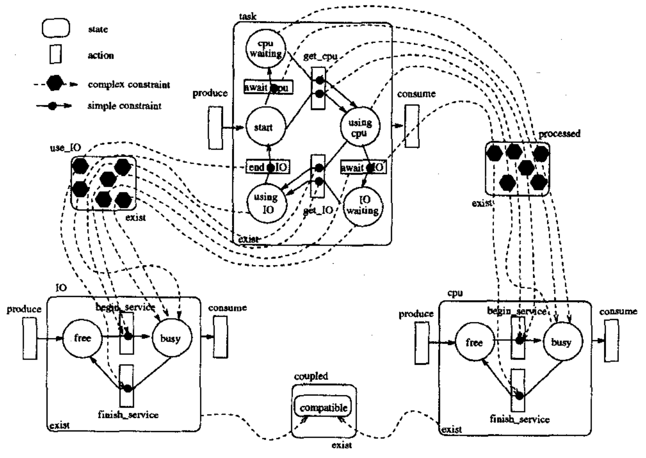

Figure 7: Behavior of a complex object.

Returning to the example, let $processed_1$ be an association instance that relates a task instance with a cpu instance. Then let $task_1$ and cpu_1 stand for the two instances of the *task* and *cpu* classes associated through $processed_1$[5]. Similarly, for the association instance use_IO_1, let $task_1$ and IO_1 stand for the two instances of the *task* and *IO* classes associated via use_IO_1. Then the description of the behavior constraints for the *task* class in the complex object is shown in Figure 7. These constraints can be put into forms as follows[6].

- γ_1 =*[task₁ :task, Act₀]* requires *state (cpu₁ :cpu, busy)*.

  Description: the $task_1$ instance should undergo a change of state from *start* to *cpu_waiting* via the activation Act_0 (that uses the function *await_cpu*) if the instance cpu_1 associated with it is in the *busy* state.

- γ_2 =*state (task₁: task,cpu_waiting)* requires *state (cpu₁: cpu, busy)*.

  Description: as long as the $task_1$ instance is in the state *cpu_waiting*, a cpu_1 instance must be in the state *busy*.

- γ_3 =*[task₁ :task, Act₁]* requires *[cpu₁ :cpu, Act₀]*.

  Description: for a *task* instance to invoke the function *get_cpu* via the activation Act_1, there must be a cpu_1 instance that is *free* and will change to the state *busy* via the activation Act_0 of transition *begin_service*. A similar constraint must be placed on the activation Act_2 of instance $task_1$.

- γ_4 =*state (task₁: task, using_cpu)* requires *state (cpu₁: cpu, busy)*.

  Description: as long as the $task_1$ instance is in the state *using_cpu*, the cpu_1 instance must be in the state *busy*.

- γ_5 =*[task₁: task, Act₃]* requires *[cpu₁: cpu, Act₁]*.

  Description: for a $task_1$ instance to invoke the function *await_IO* via the activation *[task₁ :task, Act₃]*, the cpu_1 instance that served this task has to change state from *busy* to *free* while the task is waiting for the I/O.

- γ_6 =*[task₁ :task, Act₃]* requires *state (IO₁ :IO, busy)*.

  Description: the $task_1$ instance should undergo a change of state from *using_cpu* to *IO_waiting* via the activation Act_3 (that uses the function *await_IO*) if the instance IO_1 associated with it is in the *busy* state.

- γ_7 =*state (task₁: task, IO_waiting)* requires *state (IO₁: IO, busy)*.

  Description: as long as the $task_1$ instance is in the state *IO_waiting*, the IO_1 instance must be in the state *busy*.

- γ_8 =*[task₁: task, Act₄]* requires *[IO₁: IO, Act₀]*.

  Description: for a $task_1$ instance to invoke the function *get_IO* via the activation Act_4, there must be an IO_1 instance that is *free* and will change to the state *busy* via the activation Act_0 of transition *begin_service*. A similar constraint must be placed on the activation Act_5 of instance $task_1$.

- γ_9 =*state (task₁ :task, using_IO)* requires *state (IO₁ :IO, busy)*.

  Description: as long as the $task_1$ instance is in the state *using_IO*, the IO_1 instance must be in the state *busy*.

- γ_{10} =*[task₁ :task, Act₆]* requires *[IO₁ :IO, Act₁]*.

  Description: for a $task_1$ instance to invoke the function *end_IO* via the activation *[task₁ :task, Act₆]*, the IO_1 instance that served this task has to change state from *busy* to *free* while the task is ending the use of the I/O.

The association instance $processed_1$ is constrained by the existence of an instance of the *cpu* class and a *task* class instance that are associated through this association instance. The association instance use_IO_1 is constrained by the existence of an instance of the *task* class and an *IO* class instance

[4]This general complex constraint form is more general than the specific forms used in Sakai [15]. Not all forms are needed in a typical system.

[5]We can imagine a function $[processed_1 : processed, traverse]$ that returns the instance cpu_1 of the *cpu* class that is associated with $task_1$ via the *processed* association. Also, the function $[processed_1 : processed, reverse - traverse]$ returns the $task_1$ instance associated with the cpu_1 instance.

[6]See Figures 5 and 6 for the definition of activations.

that are associated though this association instance. (Notice that we do not show these constraints in Figure 7 to reduce the complexity of the diagram.) Using the constraint forms presented above, we can write this formally as follows.

- γ_{11} =*state (processed$_1$:processed, exist)* requires *state (task$_1$:task, exist)* and *state (cpu$_1$:cpu, exist).*

  Description: as long as the *processed*$_1$ instance *exists*, an instance of *task* and an instance of *cpu* that are associated through this association instance must exist.

- γ_{12} =*state (use_IO$_1$:use_IO, exist)* requires *state (task$_1$:task, exist)* and *state(IO$_1$:IO, exist).*

  Description: as long as the *use_IO*$_1$ instance *exists*, an instance of *task* and an instance of *IO* that are associated through this association instance must exist.

- γ_{13} =*state(cpu$_1$:cpu, exist) and state(IO$_1$:IO, exist) requires state(coupled$_1$:coupled, compatible)*

  Description: as long as the two instances IO_1 and cpu_1 are coupled through and association $coupled_1$, they must meet the compatibility constraint, i.e., the speed of the *IO* is not less than 10^{-3} times the speed of the *cpu*.

Using this model of behavior, a complex object is treated as a single entity. Looked at in this way, a complex object is viewed as a *complex behavior entity* and is denoted $\Gamma(O)$, which is the set of constraints for the complex object. In the previous example, the complex behavior entity of the complex object is

$$\Gamma(execute) = \{\gamma_1, \ldots, \gamma_{13}\}.$$

A complex object O is said to be in a *valid* state if every constraint γ in the complex behavior entity $\Gamma(O)$ is either true or inapplicable. A constraint γ is *true* if both sides of the constraint are true. On the other hand, if the left hand side of the constraint is false then the constraint is said to be *inapplicable.*

3 Behavioral Views of Complex Objects

The process of defining views of complex objects may include hiding classes, adding virtual classes, altering associations, hiding associations, making new association, *etc.* From the behavioral point of view, when defining views of a complex object, one or more of the following operations is done:

- Refinement of a complex constraint.
- Removal of a complex constraint.
- Addition of a new complex constraint.
- Alteration of the *form* of a complex constraint.

We are interested in studying what effects such changes have on the behavior of complex objects and on the relationship between complex behavior before and after these changes. To do that, we must be able to define views on the behavioral level of the complex object schema. This will require the definition of algebraic operators for behavior that are similar to the concept of the class algebra introduced by Rundensteiner [13]. This can be done by defining refinement rules on our behavior model that ensure the correctness of the refinement process. The refinement process can be thought of as introducing new states and transitions, and removing states and transitions in a behavior graph. The refinement operators will be used for extracting valid and consistent sets of classes, associations and objects from a database to form a functionally correct view that obeys the complex behavior invariant and related constraints.

View correctness can have many interpretations. There are at least four different interpretations (not necessary conflicting) of view correctness in the literature: closure of a view, completeness and independence of a view, consistency of the generalization hierarchies for a view, and consistency of a view extension with the underlying database [13, 14].

A *closed* view is a view with all classes that are directly or indirectly referenced by the other classes in the view [13]. The concept of a closed view is essential for a functionally correct view, since missing classes can cause a computation to go wrong. However, this definition of closed views is too simple. Using our model the notion of closed behavior views must be defined at a behavioral level, where the use of components of a class (transitions and states) obey the closed behavior view definition. The view independence specification needs to be extended to include behavior as well as structure. This will ensure the complete semantic independence of the definition of a view.

One of the natural needs of a design evolution process is the ability of the DSVCO to handle the evolution of the underlying database of a view. Identifying the problems that the system encounters as the underlying database evolves and developing a methodology that automatically fixes these problems when possible must take behavior into account.

4 Future Directions

In this section we outline the major steps that we propose to solve the problems described in this paper. The work plan is divided into four stages: Formalization, Theoretical Analysis, Validation of the Results and Prototyping and Analysis of the Results.

The work described in this paper is a starting point for the formalization. As for the theoretical analysis, we will study the effect of the view definition algebras on the behavioral model of the complex object schema, identify points that need further analysis, develop a proper formalization of such points and present solutions. Additional operators to manipulate complex constraints in associations and classes will need to be formalized.

A limited implementation for our DSVCO to demonstrate the validity of the key parts of our work will be done. We will analyze how well our methodology solves the original problem, identify limitations, and explain the source of such limitations and how to overcome them by testing the implementation on a variety of examples.

5 Conclusion

In this paper, the problem of creating complex object views in object oriented database systems is introduced. The requirements of this problem for behavioral modeling are developed and an extension to the Life Cycle behavior model that satisfies these requirements is presented.

Future research will focus on development of rules for refining states and transitions for individual classes and constraints for associations between classes to facilitate development of views of complex objects. This will be done by developing an algebra for manipulating the behavior of the complex objects and implementing this algebra as part of the DVSCO system.

Acknowledgment: This work was partially supported by the National Science Foundation, Grant Number *DDM-9215620*. Any opinions are those of the authors.

References

[1] Serge Abiteboul and Anthony Bonner, "Object and Views", ACM SIGMOD, 1991.

[2] Amihood Amir and Nicholas Roussopoulos, "Optimal View Caching", Information Systems Vol. 15, No. 2, pp. 169-171, 1990.

[3] Jose A. Blakeley, Per-Ake Larson, and Frank Wm. Tompa, "Efficiently Updating Materialized Views", Proceeding of 1986 ACM SIGMOD International Conference on Management of Data, pp. 61-71, New York, 1986.

[4] Derek Coleman, Fiona Hayes, and Stephen Bear, 'Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design', IEEE Transaction on Software Engineering, Vol 18, No. 1, pp. 9-18, January 1992.

[5] Bogdan Czejdo and David W. Embely, "View Specification and Manipulation for A Semantic Data Model", Information Systems, Vol. 16, No. 6, pp. 585-612, 1991.

[6] Ronald Fagin, Jeffrey D. Ullman, and Moshe Y. Vardi, "On the Semantics of Updates in Databases", ACM SIGACT-SIGMOD-SIGART Symposium on Principle of Database Systems, 1983.

[7] David Harel, "Statecharts: A Visual Formalism for Complex Systems", Science of Computer Programming 8, pp. 231-274, 1987.

[8] Sandra Heiler and Stanley Zdonik, "Object Views: Extending the Vision", Sixth International Conference on Data Engineering, California, Los Angeles, Feb. 5-9 1990.

[9] Deepak Kapur and David Musser, "Tecton: A Framework for Specifying and Verifying Generic System Components", Technical Report 92-20, Computer Science Department, Rensselaer Polytechnic Institute, 1992.

[10] Deepak Kapur, "Automated Reasoning in Software Design", Technical Report, Institute for Programming and Logics, The University at Albany, New York, 1993.

[11] Kelvin W. Liu and David L. Spooner, "Object-Oriented Database Views for Supporting Multidisciplinary Concurrent Engineering", Proc. IEEE Computer Software and Applications Conf., IEEE Computer Society Press, 1993.

[12] J.L. Peterson, "Petri Net Theory and Modeling of Systems", North-Holland, 1981.

[13] Elke A. Rundensteiner, "MultiView: A Methodology for Supporting Multiple Views in Object-Oriented Database", Proceeding of the 18th VLDB Conference, Vancouver, British Columbia, 1992.

[14] Elke A Rundensteiner, "A Class Integration Algorithm and Its Application for Supporting Consistent Object Views", Technical Report 92-50, Department of Information and Computer Science, University of California, Irvine, May 1992.

[15] Hirotaka Sakai, "A Method for Contact and Delegation in Object Behavior Modeling", IEICE Tans. Inf. and Sys. Vol. E76-D, No. 6, June 1993.

[16] Bernhard Schiefer, "Supporting Integration and Evolution with Object-Oriented Views", FZI-Report 15/93, Forschungszentrum Informatik (FZI), Germany, July 1993.

[17] Michael Schrefl, "Behavior Modeling by Stepwise Refining Behavior Diagrams", Entity-Relationship Approach: The Core of Conceptual Modeling, Proc. of the 9th Int. Conf. on the Entity-Relationship Approach, Lausanne, Switzerland, 8-10 Oct., 1990.

[18] J.J. Shilling and P.F. Sweeney, "Three steps to views: Extending the object-oriented paradigm", Proceeding of the International Conference on Object-Oriented Programming, pp. 353-361, 1989.

[19] J.M. Spivey, "Understanding Z, a Specification Language and its Formal Semantics", Readings, Cambridge University Press, 1988.

[20] D. Spooner and M. Hardwick, "Using Persistent Object Technology to Support Concurrent Engineering", Concurrent Engineering, editors P. Gu and A. Kusiak, Elsevier Publishing Company, 1993.

[21] The STEP Programmer's Tool Kit, Reference Manual Version 1.1, STEP Tools, Inc, Rensselaer Technology Park, Troy, NY, 1992.

[22] Kazuyuki Tsuda, and Kensaku Yamamoto, Masahito Hirakawa, Minoru Tanaka, and Tadao Ichikawa, " MORE: An Object-Oriented Data Model with a Facility for Changing Object Structures", IEEE Transactions on Knowledge and Data Engineering, Vol. 3, No. 4, pp. 444-460, Dec. 1991.