CONTRAINT ANALYSIS FOR SPECIFYING PERSPECTIVES OF CLASS OBJECTS

Susan Dariing Urban †
Lois M. L. Delcambre ‡
† Department of Electrical and Computer Engineering
University of Miami
Coral Gables, Florida 33124
‡ The Center For Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, Louisiana 70504

Abstract

This paper presents an approach known as constraint analysis for specifying updatable perspectives of class objects in an object-oriented data schema. Perspectives provide a way of defining the scope of a user's view of a schema. Operations on class objects are then defined in terms of specific perspectives based on the semantics associated with the schema. Schema semantics are represented as a set of integrity constraints expressed in Horn logic. The constraint analysis process then reasons about schema constraints to support a flexible approach to update propagation. The advantage of constraint analysis is that both inherent and explicit constraints can be used to support the automated specification of updatable perspectives that preserve object integrity.

1 Introduction

User views of class objects are often neglected in object-oriented data modeling. Class definitions are typically viewed as being globally available to all users. Furthermore, subclasses freely inherit properties and operations of superclasses, often with the capability of further restricting inherited property values or extending and modifying inherited operations. Bloom [BlooS7] describes this global accessibility of class objects as a potential threat to data abstraction, where an application can gain access to the representation of an object through the instance variables of a class. This global accessibility of class objects also ignores the fact that some users do not need to see the entire schema and that different users need to view class objects in different ways.

Forming views of class objects, however, is not a straightforward task. As described in [Urba87a], constraints of the schema together with operations on objects and security restrictions can cause difficulty for the correct formation of user views. Views of class objects should restrict access to selected properties and to selected objects that can be reached through functional composition on object properties. Schema constraints, however, should not be violated because of restricted views or because a view is unaware of the constraints imposed by a different user view. Ideally, views and operations on views should be formed by considering the effects of schema constraints on object manipulation to determine that 1) the appropriate properties and objects are included in each view, and 2) update propagation actions do not violate schema constraints or security restrictions. This approach to forming views is particularly important in design applications where each design group has its own set of constraints that must not be violated by other design groups.

This paper presents an approach known as constraint analysis [Urba87b] for forming views of class objects, demonstrating how schema constraints expressed in first-order logic can be analyzed to support the automated specification of user views that protect object integrity. The primary contribution of this logic-based approach to forming user views is that schema constraints, including explicit constraints that are declaratively expressed in a constraint language, are uniformly represented in a formal, analyzable manner. As a result, when a designer constructs an updatable view of a class, the uniform set of schema constraints can be analyzed to insure that

1. the appropriate properties are included in the view based on the operations to be performed,
2. the appropriate operations are propagated to insure that schema constraints are satisfied, and
3. the full scope of a view is considered (i.e., additional objects are identified that may need to be viewed in the course of satisfying schema constraints).

Constraint analysis therefore provides a formal basis for insuring that views and operations on views are correctly specified. Furthermore, analyzing schema constraints to understand the ways in which constraints can be violated and the ways in which constraints can be satisfied supports a more creative and flexible environment for specifying propagation actions. Different views can therefore be tailored to the needs and/or restrictions of each user group. The fact that explicit constraints can be included in the specification process is particularly important since previous approaches to update propagation rarely address the effects of explicit constraints on the propagation process [Hect81, Brod84, Abir85, Moga84, Mark86].

The view mechanism used in this research is known as perspectives [Urba87a]. A perspective is the window through which a user sees an abstract object. The primary purpose of this paper is to describe the application of constraint analysis to the formation of class perspectives. A brief discussion of related work is first presented in Section 2 followed by an overview of perspectives in Section 3. The representation of schema constraints in first-order logic is then presented in Section 4. An example of forming perspectives by constraint analysis is presented in Section 5. The paper concludes with a summary and discussion of future research in Section 6.
2. Related Work

Constraint analysis has been influenced by semantic modeling research on design methodologies [Mylo80, King84, Brod84] and update propagation [Abit85] and also by research on constraint-based systems [Shep84, Morg84]. The design methodology most relevant to the current research is the Active and Passive Component Modeling methodology (ACM/PCM) associated with the SHM+ semantic data model [Brod84] where data operations are specified by considering the forms of abstraction supported by SHM+. The research presented in this paper represents an extended application of ACM/PCM where database operations are formed by also considering the effects of explicit constraints. Furthermore, in the research presented here, all constraints of the schema are automatically managed so that the constraints to be addressed are explicitly identified.

The research in [Brod84] is essentially a design approach based on update propagation. Previous work on update propagation includes the work of Hecht and Kerschberg [Hect81], Abiteboul and Hull [Abit85], and Mark and Roussopoulos [Mark86]. Each of the above research efforts presents fixed update algorithms for inherent schema constraints. As with the SHM+ design methodology, the propagation algorithms do not include the analysis of explicit interclass constraints and they do not consider the fact that update propagation may differ depending on different user views.

Another related area of work is the research of Nicolas and Yazdanian on integrity checking in deductive databases [Nico78]. As described in [Nico78], by transforming constraints expressed in first-order logic into conjunctive normal form, it is easier to reason about the effects of low-level database operations and to analyze the alternatives that can be taken to satisfy constraints when constraint violations occur. Constraint analysis represents an extension of the work in [Nico78] as described in [Urba88], as well as a different application of the reasoning process.

3. Perspectives of Class Objects

Perspectives can be described as non-first normal form relations with basic update operations defined in terms of schema semantics. The basic operations are 1) the ADD operation to add an object as an instance of a class, 2) the REMOVE operation to remove an object as an instance of a class, 3) the INSERT operation to insert an object as a property value of another object, and 4) the DELETE operation to delete an object as a property value of another object. These four operations directly manipulate objects in the database and, as a result, can affect object integrity. High level transactions are defined on top of these low-level operations [Brod84]. The perspectives of a particular user group are collected into a user view to support the specification of transactions.

Three types of perspectives exist within a user view: primary perspectives, nested perspectives, and dependent perspectives. To present each perspective type, consider a schema of the SHIPS database [Hamm81] in Figure 1. The data model used to describe the schema is a synthesis of existing semantic and object-oriented data models [Hull87]. In the graphical notation of Figure 1, circles represent abstract objects while rectangles represent simple, printable objects. Unlabelled arcs represent ISA connections while labelled arcs represent property definitions. Double-headed arrows represent multi-valued properties. A solid arrow head defines a required property that does not allow null values. A property definition with arrows at both ends is an inverse property definition.

Primary perspectives are formed for classes that represent the objects of primary interest in a view. For example, the following is a primary perspective of the SHIP class:

\[ \text{SHIP:} \{ \text{Name, Hull#, Type, HomePort, \{CargoType\}} \]

In the above perspective, CargoType is enclosed in braces to indicate that it is a multi-valued property.

A primary perspective can include properties from the perspective class (i.e., the class over which the perspective is formed), as well as inherited properties. Properties not included in a perspective represent properties that are not accessible in the view. As another example, a different user may need the following primary perspective of oil tankers:

\[ \text{OIL-TANKER:} \{ \text{Name, Hull#, CountryOfRegistry, \{Inspections\}, #OfInspections, DateLastInspection} \]

The OIL-TANKER perspective includes properties defined for the OIL-TANKER class as well as properties inherited from the SHIP class. Properties of SHIP and OIL-TANKER objects, such as HomePort, CargoType, and HullType are not accessible in this particular perspective.

The second type of perspective is a nested perspective. Nested perspectives must be formed for all properties of a given perspective that have abstract objects as property values. For example, in the OIL-TANKER perspective, the CountryOfRegistry and Inspections properties have abstract objects as values from the COUNTRY and INSPECTION classes, respectively. Nested perspectives describe how much of the COUNTRY and INSPECTION classes can be seen by the OIL-TANKER perspective:

\[ \text{OIL-TANKER.CountryOfRegistry: CountryName} \]
\[ \text{OIL-TANKER.Inspections: InspectionDate} \]

Dependent perspectives represent a final type of perspective. As an example, consider again the primary perspective of the SHIP class along with the explicit constraints in Figure 2. Constraint 1 states that all instances of the SHIP class having "oil" as a cargo type must also be instances of the OIL-TANKER class. If a user of the SHIP perspective is allowed to modify the CargoType property, inserting a cargo type of "oil" for a SHIP object will require that the object also be added to the OIL-TANKER class. A dependent perspective of the OIL-TANKER class is needed by the SHIP perspective so that the appropriate information can be entered about oil tanker objects:

\[ \text{SHIP/OIL-TANKER:} \text{HullType} \]

The examples above present perspectives as a way of defining views of class objects. Each primary perspective corresponds to a limited view of one class with optional nested and dependent perspectives for additional access to properties of abstract object-valued attributes and to subclasses of the primary perspective class. Both nested and dependent perspectives can have other nested and dependent perspectives. A perspective must be formed, however, by considering the operations to be performed on objects in the perspective and the constraints of the schema. The following sections describe how inherent constraints together with additional explicit constraints can be used to support the specification of class perspectives.

4. Schema Constraints in First-Order Logic

To generate an analyzable set of constraints that describe the semantics of a schema, both the inherent and explicit constraints
of the schema are described in first-order logic and then transformed into conjunctive normal form. Explicit constraints can be directly translated to first-order logic since the constraint language used in this research is a user-oriented form of first-order logic [Urba87b]. The first-order logic form of constraints captured directly in the schema description is based on the logical definition of each model construct, as in the definition of RML [Gree84]. Each logical definition of a model construct is then "instantiated" for every occurrence of that construct in an application schema. This instantiation process produces a set of constraints that describes the semantics of a specific schema.

As an example, the following "generic" constraint defines the meaning of a superclass/subclass (i.e., ISA) relationship:

\[ \text{isa}(C_1, C_2) \land \text{instance}(C_1, O) \rightarrow \text{instance}(C_2, O). \]

The above constraint states that if \( C_1 \) is a subclass of \( C_2 \) and \( O \) is an instance of \( C_1 \), then \( O \) must also be an instance of \( C_2 \). This constraint is instantiated for the schema in Figure 1 to produce the ISA constraints shown in Figure 3.

In addition to ISA constraints, other "generic" constraints are presented in [Urba87b] that define the meaning of: property induction [Mylo80] (i.e., the valid set of values for a property of an object), required properties, single-valued properties, unique (1-1) properties, inverse properties, disjoint categories, and keys. The constraints describing the schema of Figure 1 are shown in Figure 3. Each constraint is expressed in Horn clause form to support the analysis process described in the following section.

5. Forming Perspectives By Constraint Analysis

The advantage of expressing schema constraints in conjunctive normal form is that it is easier to reason about the effects of constraints on the manipulation of database objects [Nico78]. To illustrate the reasoning process, consider a set of schema constraints in conjunctive normal form:

\[ C_1 \land C_2 \land \cdots \land C_m. \]

Each clause \( C_l \) can be written as an implication \( p_1 \land p_2 \land \cdots \land p_n \rightarrow q_1 \lor q_2 \lor \cdots \lor q_m \), where \( n \geq 1 \) and, for the purpose of this research, \( 0 \leq m \leq 1 \) (i.e., each clause is a Horn clause).

In order for schema constraints to be satisfied, each clause in the conjunction of clauses must be satisfied. Each clause, however, can evaluate to true in different ways based on the truth value for \( P \rightarrow Q \). For example, when \( P \) is true, \( Q \) must be true or an operation must be invoked to make \( Q \) true. Likewise, if an operation causes \( Q \) to become false, then either \( Q \) must be
ISA CONSTRAINTS:
1: OIL-TANKER(S) → SHIP(S).
2: MERCHANT-SHIP(S) → SHIP(S).
3: BANNED-SHIP(S) → SHIP(S).
4: BANNED-OIL-TANKER(S) → BANNED-SHIP(S).
5: BANNED-OIL-TANKER(S) → OIL-TANKER(S).

PROPERTY INDUCTION CONSTRAINTS:
6: SHIP(S) & Name(S,O) → STRING(O).
7: SHIP(S) & Hull#(S,O) → INTEGER(O).
8: SHIP(S) & Type(S,O) → SHIP-TYPE(O).
9: SHIP(S) & HomePort(S,O) → CITY-TYPE(O).
10: SHIP(S) & CargoType(S,O) → CARGOTYPE-TYPE(O).
11: SHIP(S) & ShipEngines(S,O) → ENGINE(O).
12: SHIP(S) & Captain(S,O) → ASSIGNMENT(O).
13: SHIP(S) & CountryOfRegistry(S,O) → COUNTRY(S,O).
14: OIL-TANKER(S) & HullType(S,O) → HULLTYPE-TYPE(O).
15: OIL-TANKER(S) & Inspections(S,O) → INSPECTION(O).
16: OIL-TANKER(S) & #OfInspections(S,O) → INTEGER(O).
17: OIL-TANKER(S) & DateLastInspection(S,O) → DATE-TYPE(O).
18: COUNTRY(S) & ShipsRegisteredHere(S,O) → SHIPS(S).
19: COUNTRY(S) & CountryName(S,O) → STRING(O).
20: ENGINE(S) & SerialNumber(S,O) → STRING(O).
21: ENGINE(S) & KindOfEngine(S,O) → ENGINE-TYPE(O).
22: OFFICER(S) & OfficerName(S,O) → STRING(O).
23: OFFICER(S) & DateCommissioned(S,O) → DATE-TYPE(O).
24: ASSIGNMENT(S) & OfficerAssigned(S,O) → OFFICER(O).
25: ASSIGNMENT(S) & ShipAssignedTo(S,O) → SHIPS(S).
26: ASSIGNMENT(S) & OfficerAssigned(S,O) → OFFICER(O).
27: ASSIGNMENT(S) & DateAssigned(S,O) → DATE-TYPE(O).

REQUIRED PROPERTY CONSTRAINTS:
28: SHIP(S) → Name(S,fa(S)).
29: SHIP(S) → Hull#(S,fb(S)).
30: SHIP(S) → CargoType(S,fc(S)).
31: SHIP(S) → CountryOfRegistry(S,fd(S)).
32: SHIP(S) → HomePort(S,fe(S)).
33: SHIP(S) → Type(S,ff(S)).
34: SHIP(S) → Name(S,fg(S)).
35: SHIP(S) → HomePort(S,fh(S)).
36: SHIP(S) → CountryName(S,fi(S)).
37: ENGINE(S) → SerialNumber(S,fj(S)).
38: OFFICER(S) → OfficerName(S,fl(S)).
39: OFFICER(S) → DateCommissioned(S,fm(S)).
40: ASSIGNMENT(S) → OfficerAssigned(S,fn(S)).
41: ASSIGNMENT(S) → ShipAssignedTo(S,fo(S)).
42: COUNTRY(S) → CountryName(S,fp(S)).
43: ENGINE(S) → SerialNumber(S,fq(S)).
44: OFFICER(S) → OfficerName(S,fr(S)).
45: OFFICER(S) → DateCommissioned(S,fs(S)).
46: ASSIGNMENT(S) → OfficerAssigned(S,ft(S)).
47: ASSIGNMENT(S) → ShipAssignedTo(S,fu(S)).

SINGLE-VALUED PROPERTY CONSTRAINTS:
48: SHIP(S) & Name(S,O1) & Name(S,O2) & not.equal(O1,O2) →.
49: SHIP(S) & Hull#(S,O1) & Hull#(S,O2) & not.equal(O1,O2) →.
50: SHIP(S) & Type(S,O1) & Type(S,O2) & not.equal(O1,O2) →.
51: SHIP(S) & HomePort(S,O1) & HomePort(S,O2) & not.equal(O1,O2) →.
52: SHIP(S) & Shirt(S,O1) & Shirt(S,O2) & not.equal(O1,O2) →.
53: OIL-TANKER(S) & #Inspections(S,O1) & #Inspections(S,O2) & not.equal(O1,O2) →.
54: OIL-TANKER(S) & DateLastInspection(S,O1) & DateLastInspection(S,O2) & not.equal(O1,O2) →.
55: OIL-TANKER(S) & HullType(S,O1) & HullType(S,O2) & not.equal(O1,O2) →.
56: OFFICER(S) & OfficerName(S,O1) & OfficerName(S,O2) & not.equal(O1,O2) →.
57: OFFICER(S) & DateCommissioned(S,O1) & DateCommissioned(S,O2) & not.equal(O1,O2) →.
58: ASSIGNMENT(S) & OfficerAssigned(S,O1) & OfficerAssigned(S,O2) & not.equal(O1,O2) →.
59: ASSIGNMENT(S) & ShipAssignedTo(S,O1) & ShipAssignedTo(S,O2) & not.equal(O1,O2) →.

Figure 3: Schema Constraints for SHIPS Database
SINGLE-VALUED PROPERTY CONSTRAINTS (Continued):
50: ASSIGNMENT(S) & DateAssigned(S,01) & DateAssigned(S,02) & not_equal(01,O2) →.
51: COUNTRY(S) & CountryName(S,01) & CountryName(S,02) & not_equal(01,O2) →.
52: ENGINE(S) & SerialNumber(S,01) & SerialNumber(S,02) & not_equal(01,O2) →.
53: ENGINE(S) & KindOfEngine(S,01) & KindOfEngine(S,02) & not_equal(01,O2) →.

INVERSE PROPERTY CONSTRAINTS:
54: SHIP(S) & CountryOfRegistry(S,O) → ShipsRegisteredHere(O,S).
55: COUNTRY(S) & ShipsRegisteredHere(O,S) → CountryOfRegistry(O,S).
56: SHIP(S) & Captain(S,O) → ShipAssignedTo(O,S).
57: ASSIGNMENT(S) & ShipAssignedTo(S,O) → Captain(O,S).

KEY CONSTRAINTS:
58: SHIP(S1) & SHIP(S2) & Name(S1,01) & Name(S2,01) & Hull#(S1,02) & Hull#(S2,02) → equal(S1,S2).

EXPLICIT CONSTRAINTS:
59: SHIP(S) & CargoType(S,O) & equal(O,"oil") → OIL-TANKER(S).
60: SHIP(S) & Type(S,O) & equal(O,"merchant") → MERCHANT-SHIP(S).

Figure 3: Schema Constraints for SHIPS Database - Continued

reasoning process can be used to identify the classes that affect a given operation and the alternative actions to invoke when constraint violations occur.

Since ADD and INSERT represent operations that make conditions about data objects true, the clauses affecting an ADD or an INSERT operation are those clauses where the class or property involved in the operation appears on the left hand side of the clause [Nico78]. Similarly, since REMOVE and DELETE represent operations that make conditions about data objects false, the clauses affecting REMOVE or DELETE are those clauses where the class or property of the operation appears on the right hand side of the clause [Nico78].

When analyzing an ADD/INSERT operation, each clause affecting the operation is identified. For each clause, if the remaining conditions on the left hand side of the clause are true at execution time, the condition on the right hand side of the clause must also be true or it must be made true. The ADD/INSERT operation will abort if the condition is not true. As alternative options, the designer can decide to: 1) invoke an operation to make the condition on the right hand side of the clause true, or 2) invoke an operation to falsify a condition on the left hand side. The alternative options seek to actively satisfy constraints when constraint violations occur. Each propagation action must be recursively analyzed against the schema constraints.

For a REMOVE/DELETE operation, the clauses affecting the operation are first identified. For each clause, if all conditions on the left hand side of the clause are true when the delete operation occurs, the designer can choose to: 1) abort the delete operation, 2) invoke the appropriate operations to reestablish the right hand side of the clause, or 3) invoke the appropriate operations to falsify a condition on the left hand side of the clause. Options 2 and 3 initiate propagation actions that must be recursively analyzed against the schema constraints.

The complete representation and analysis of constraints is more complex than described above. Since the primary purpose of this paper is to describe the utility of constraint analysis for forming updatable user views, the analysis process is presented in its simplest form, where each inherent and explicit constraint can be expressed as a single Horn clause. A more detailed presentation of constraint analysis can be found in [Urban76, Urban88].

To illustrate how constraint analysis can be used to specify updatable perspectives of class objects, consider forming a primary perspective of the SHIP class. The following operations will be allowed on the perspective: ADD, REMOVE, and INSERT/DELETE on all properties of the perspective.

To specify the ADD operation, all clauses where "SHIP" occurs on the left hand side are identified. Referring to Figure 3, the clauses found are 6-13 (property induction constraints), 28-31 (required property constraints), 38-42 (single-valued property constraints), 54 and 56 (inverse property constraints), 58 (a key constraint), and 59-60 (explicit constraints). All clauses except 28-31 can be immediately eliminated from the analysis process since the left hand side of each clause defines conditions on properties of the SHIPS class. Since an object will not have property values when the object is initially added to a class, the left hand side of such clauses can be assumed false at the time of the ADD operation. These clauses will later be analyzed in the context of INSERT operations on specific properties. The remaining clauses (28-31) will directly affect the ADD operation:
28: SHIP(S) → Name(S,Oa(S)).
29: SHIP(S) → Hull#(S,Cb(S)).
30: SHIP(S) → CargoType(S,fc(S)).
31: SHIP(S) → CountryOfRegistry(S,fd(S)).

All of the above clauses define required properties, where variables beginning with "^" represent Skolem functions. As a result, Name, Hull#, CargoType, and CountryOfRegistry must be included in a perspective of the SHIP class if the ADD operation is to be allowed. All other properties of the SHIP class can be optionally included if they are needed in the perspective. Furthermore, the ADD operation must be accompanied by INSERT operations on each property listed above for the SHIP object to be added. If the designer decides to include all properties of the SHIP class in the perspective, the primary perspective is:

{SHIP: Name, Hull#, Type, HomePort, {CargoType}, {ShipEngines}, CountryOfRegistry, Captain}
SHIP.ADD(O)
begin
assert SHIP(O)
exception:
if not Name(O,N) then
get(N); INSERT Name(O,N) {* Specification of INSERT on Name goes here *}
if not Hull#(O,H) then
get(H); INSERT Hull#(O,H) {* Specification of INSERT on Hull# goes here *}
if not CargoType(O,C) then
get(C); INSERT CargoType(O,C) {* Specification of INSERT on CargoType goes here *}
if not CountryOfRegistry(O,R) then
get(R); INSERT CountryOfRegistry(O,R)
{* Specification of INSERT on CountryOfRegistry goes here *}
end ADD

Figure 4: Initial Specification of ADD Operation on the SHIP perspective

The specification for the ADD operation on the SHIP primary perspective at this point in the analysis process is shown in Figure 4. The assert operation is a primitive database operation that makes an object an instance of a class. The exception part following the assert operation provides a way of automatically invoking the operations that preserve object integrity. Each condition on the right hand side of the four required property constraints appears as an exception condition. Each get operation is assumed to "get" the values to be inserted from an input buffer where the user has already entered property values for the SHIP perspective. In the case of abstract-object-valued properties, such as CountryOfRegistry objects, the user enters a value for the identifying properties, such as CountryName, which are used by the get operation to identify the object to be related to the SHIP object.

It is important to understand that the code in Figure 4 specifies what propagation actions will occur and not necessarily the order in which propagation actions should happen. Current research on constraint enforcement using the constraint analysis process is investigating the timing aspects of propagation actions. After analyzing the clauses that affect the ADD operation, each INSERT operation in Figure 4 must then be specified. As an example, consider the INSERT operation on CargoType. The clauses affecting the operation are:

10: SHIP(S) & CargoType(S,O) → CARGOTYPE-TYPE(O).
59: SHIP(S) & CargoType(S,O) & Equal(O,"oil") → OIL-TANKER(S).

Clause 10 simply defines the valid set of values for a CargoType property. This example assumes that the condition on the right hand side of clause 10 will be true at execution time. Otherwise the INSERT operation on CargoType will not be allowed. Clause 59, however, states that if the CargoType is "oil", then the SHIP object must also be an instance of the OIL-TANKER class. As a result, this particular perspective of SHIP needs a dependent perspective of the OIL-TANKER class with an ADD operation in order to satisfy clause 59. If the dependent perspective is not defined, then the INSERT operation on CargoType property of the SHIP perspective is either not allowed or is allowed but will abort when inserting a cargo type of "oil".

When analyzing the ADD operation on the dependent perspective of OIL-TANKER, the following clauses must be used to specify the operation:

1: OIL-TANKER(S) → SHIP(S).
32: OIL-TANKER(S) → HullType(S,fa(S)).

Clause 1 and clause 32 state that two conditions must be satisfied when S is inserted as an oil tanker: S must be an object in the SHIP class and S must have a HullType value. Since S will already be an instance of the SHIP class, the right hand side of clause 1 can be assumed true at execution time. Clause 32 states that HullType is a required property that must be included in the dependent perspective of OIL-TANKER if an ADD operation is to be allowed. Furthermore, the ADD operation on OIL-TANKER must be accompanied by an INSERT operation on the HullType property.

The specification of the ADD operation on the SHIPs primary perspective can now be refined as shown in Figure 5. The conditions that are assumed to be true at execution time, such as the right hand sides of clauses 10 and 32, are not listed in the exceptions section. If those conditions are violated at execution time, the ADD operation will abort since no propagation action is specified. Only the conditions listed in the exceptions part will initiate propagation actions. A different perspective of the SHIP class may list different conditions and different propagation actions in the exceptions part of the specification. The constraint analysis process therefore supports flexibility in the specification of propagation actions.

The example presented so far has established a need for a primary perspective of the SHIP class with required and optional properties and a dependent perspective of the OIL-TANKER class with HullType as a required property. Nested perspectives can be formed using the same process described above. The SHIP perspective and the OIL-TANKER perspective together with nested perspectives of abstract objects are shown below:

{SHIP: Name, Hull#, Type, HomePort, {CargoType},
{ShipEngines}, CountryOfRegistry, Captain}
{SHIP.ShipEngines: SerialNumber}
{SHIP.CountryOfRegistry: CountryName}
{SHIP.Captain.OfficerAssigned: OfficerName}
{SHIP/OIL-TANKER: HullType}

The specification process is not complete until all propagation actions are analyzed against the constraints of the schema.
Rather than present the complete specification for the ADD operation on the SHIP perspective, the remainder of this example describes how the schema constraints can be used to specify a REMOVE operation.

To specify a REMOVE operation for the SHIP perspective, all clauses where "SHIP" occurs on the right hand side must be identified. Those clauses are:

1: OIL-TANKER(S) \rightarrow \text{SHIP}(S).
2: MERCHANT-SHIP(S) \rightarrow \text{SHIP}(S).
3: BANNED-SHIP(S) \rightarrow \text{SHIP}(S).
19: COUNTRY(S) \& \text{ShipsRegisteredHere}(S,O) \rightarrow \text{SHIP}(O).
26: ASSIGNMENT(S) \& \text{ShipAssignedTo}(S,O) \rightarrow \text{SHIP}(O).

Clauses 1-3 are ISA constraints. As a result, if the right hand side of each clause is falsified, the left hand side of each clause must also be falsified. Stated in different terms, an object deleted from a superclass must also be deleted from its subclasses. The remaining clauses (clauses 19 and 26) indicate that references to the deleted SHIP object must be removed and, as a result, DELETE operations must be invoked on ShipsRegisteredHere and ShipAssignedTo. A partial specification of the REMOVE operation is shown in Figure 6. As with the ADD operation, the specification process is not complete until all propagation actions are analyzed against the schema constraints. For example, analyzing the DELETE operation on ShipAssignedTo will discover that ShipAssignedTo is a required property of the ASSIGNMENT class. Additional propagation actions will need to be invoked to either remove the assignment object as an instance of the ASSIGNMENT class or to insert a new SHIP object as the value of ShipAssignedTo (i.e., assign the officer of the removed SHIP object as captain of a different ship).

6. Summary

This paper has presented an approach known as constraint analysis as a means for specifying updatable perspectives of class objects in an object-oriented data schema. Perspectives provide a way of defining the scope of a user's view of the schema. Operations on class objects are then defined in terms of specific perspectives. Forming perspectives and specifying operations on perspectives, however, depends on knowledge about schema constraints. Constraint analysis supports the specification of updatable perspectives by reasoning about inherent and explicit constraints that are expressed as a set of Horn clauses. The ability to analyze constraints represented as Horn clauses helps the designer understand the constraint violations that can occur at execution time as well as the alternative propagation actions that can be invoked when violations occur. In large applications, a formal, automated approach such as constraint analysis may be required to help a designer manage the constraints that must be considered in the specification of object manipulation operations.

Future research is proceeding in two different directions. One direction is related to constraint enforcement. An object-oriented database system with a constraint enforcement subsystem based on constraint analysis is currently under investigation. Another research direction involves the development of a constraint explanation tool. The explanation tool will be part of a prototyping environment in which the constraint analysis process is used to automatically generate code to test the functionality of object-oriented database applications.

References

SHIP.REMOVE(0)
begin
  retract SHIP(0)
except ions:
  if OIL-TANKER(0) then
    OIL-TANKER.REMOVE(0)
    begin
      retract OIL-TANKER(0)
      exceptions:
        if BANNED-OIL-TANKER(0) then
          BANNED-OIL-TANKER.REMOVE(0)
      end REMOVE
  end REMOVE
  if MERCHANT-SHIP(0) then
    if BANNED-SHIP(0) then
      if ShipsRegisteredHere(S,O) then
        if ShipAssignedTo(T,O) then
          MERCHANT-SHIP.REMOVE(0)
          {*
            Specification of REMOVE goes here
          *}
        BANNED-SHIP.REMOVE(0)
        {*
          Specification of REMOVE goes here
        *}
        DELETE ShipsRegisteredHere(S,O)
        {*
          Specification of DELETE goes here
        *}
        DELETE ShipAssignedTo(T,O)
        {*
          Specification of DELETE goes here
        *}
end

Figure 6: Partial Specification of REMOVE Operation on the SHIP perspective


