Extracting Concurrency from Objects: A Methodology

Panos K. Chrysanthis *S. Raghuram* *Krithi Ramamritham*

Department of Computer and Information Science
University of Massachusetts
Amherst, MA. 01003
e-mail: {panos, raghuram, krithi}@cs.umass.edu

Abstract

Whereas a number of semantics-based concurrency control schemes for object-oriented systems have been proposed in the literature, each scheme has approached the issue from fairly narrow considerations. In this paper, we have made an effort to discover, from first principles, the nature of concurrency semantics inherent in objects. Towards this end, we identify the dimensions along which object and operation semantics can be modeled. These dimensions are then used to classify and unify existing semantic-based concurrency control schemes. To formalize this classification, we propose a graph representation for objects that can be derived from the abstract specification of an object. Based on this representation, which helps to identify the semantic information inherent in an object, we propose a methodology that shows how various semantic notions applicable to concurrency control can be effectively combined to improve concurrency. In this process, we identify and exploit a new source of semantic information, namely, the ordering among component objects, to further enhance concurrency. Lastly, we present a scheme, based on this methodology, for deriving *compatibility tables for operations on objects.*

1 Introduction

In order to capture the needs of emerging information-intensive applications such as CAD/CAM, office information systems, and stock trading databases, several extensions to the traditional data and transaction models have been proposed [14]. For example, instead of the read/write model of data, an abstract data type model has been advocated to capture the data in complex databases. Abstract data types being a rich source of semantic information, allow the design of *type-specific* concurrency control schemes which enhance concurrency within objects, i.e., instances of abstract data types. These schemes exploit the semantic information about the types and their operations. Several forms of type-specific concurrency control techniques have been reported in the literature [3, 8, 13, 15] and the improved performance achieved by these schemes has already been demonstrated [3].

Our goal in this paper is to examine whether there is a systematic way to extract and exploit the concurrency inherent in an object. To address this issue, we consider the following to be prerequisites: (1) A precise model for objects that will bring out their concurrency semantics. (2) Given this, a scheme for extracting information that can be used for controlling concurrent access to these objects.

In Section 2, the issues related to semantic-based concurrency control are examined. In section 3, existing semantic notions are presented and characterized in terms of the different semantic information classes identified in Section 2. Section 4 presents a model that captures the abstract structure of an object and provides a way to specify the effect of an operation on an object in terms of such an abstract structure. Using this, in Section 5, a methodology is developed for deriving the compatibility table for objects. Section 6 concludes with a summary and future steps.

2 Semantics-Based Concurrency Control

In a database modeled in terms of objects, i.e., instances of abstract data types, transactions invoke operations defined on the objects. Controlling the concurrent execution of these transactions involves the control of execution of the operations invoked on the objects. Whether or not two operations, invoked by different transactions, can be allowed to execute concurrently depends on the *effect of one operation on the other*, and the *effect of the operations on the object.*

In this section, we elaborate upon these effects in order to set the stage for a characterization of semantics-based concurrency control schemes in subsequent sections. Throughout this paper, we assume a traditional transaction model in which transactions have the properties of *serializability* and *failure atomicity* [6, 7, 4].

We will be using the QStack object described below to illustrate various concepts discussed in the rest of the paper. A QStack combines the properties of a stack and a queue. The operations defined on a QStack are:

Enq(e): ok/nok or Push(e): ok/nok adds an element *e* to the back of the QStack. It returns *ok* if QStack is not full, *nok* (denoting *overflow*) otherwise.

Deq(): e/nok deletes an element *e* from the front of the QStack. It returns *e* if QStack is not empty, *nok* (denoting *empty*) otherwise.

This material is based upon work supported by the National Science Foundation under grant DCR-8500332.

We prefer to use the more general terms observer and modifier, rather than read and write, to explicitly denote the fact that a modifier may not "write" into the complete object and an observer may only "read" part of the object. Thus, as we shall see in Section 4, it may be possible for two modifiers or even a modifier and an observer to concurrently access an object.

The above classification of operations forms the basis for the methodology outlined in Section 5 for extracting concurrency from objects.

2.2 Effects of Operations on Objects

We now turn our attention to the effects of individual operations on objects. Broadly speaking, the concurrency semantics of an object can be extracted from the following:

1. semantics of the operations,
2. operation input/output values,
3. organization of the object, and
4. object usage.

Operation semantics are the most commonly used semantics in the context of concurrency control and are related to the effects of an operation on the state of an object. As we just saw, operations can be broadly classified as *observers*, *modifiers* or *modifier-observers*. Reads and writes are simple examples of observers and modifiers respectively.

Input/output semantics refer to both the direction (in/out) of information flow from an object, and to the interpretation of input and output values of an operation. The information into or out of an object occurs via the arguments of the operations defined on the object and through the outcome and results of the operations. For example, an operation without arguments such as Deq on a QStack, does not support information flow into the QStack although it supports information flow out of QStack. Interpretation of input/output values can be used to decide if two operations conflict. For example, two Push operations which attempt to push the same item onto a stack commute and thus, they do not conflict even though, in general, two Push operations conflict.

Object organization semantics refer to the abstract organization of an object. We classify this further as *composition* semantics that pertain to what an object is composed of, and as *order* semantics that refer to the relative ordering among the component objects.

Usage semantics refer to how the object is used and what is done with the information extracted out of an object by an operation. In this paper, we don't consider usage semantic information, although such information can potentially be exploited to enhance concurrency within a given application. We return to this in the concluding section.

Several techniques have been proposed in the literature to enhance concurrent access to objects. We review them in Section 3 and show how they are designed to use operation semantics, input/output semantics and object organization semantics.

3 Characterization of Object-based Concurrency Control Schemes

Commutativity is the traditional semantic notion used to determine if two operations can be allowed to execute concurrently (e.g., two reads commute). Commutativity does not distinguish between abort-dependencies and commit-dependencies. Two operations do not commute if either type of dependency may result if they execute concurrently.

Several concurrency control schemes use input/output data semantics, operation semantics, and object organization semantics in determining commuting operations [1, 12]. In [15], commutativity is defined in terms of state machines as *forward commutativity*, which is applicable only with intentions lists based recovery, and *backward commutativity*, which is applicable only with log based recovery. Multilevel concurrency-control takes object organization semantics into account [10].

An alternative method for defining conflicts is based on *serial dependency relations* [8]. An operation o_1 conflicts with another operation o_2 according to a serial dependency relation if o_1 can invalidate o_2 by appearing earlier in a serial sequence. Specifically, if there exist operation sequences h_1 and h_2 such that $h_1 \cdot o_2 \cdot h_2$ and $o_1 \cdot h_1 \cdot h_2$ are legal sequences, but $o_1 \cdot h_1 \cdot o_2 \cdot h_2$ is not, then o_1 *invalidates* o_2 and o_2 has a serial dependency on o_1. This criterion is feasible only if intentions lists based recovery is used. The use of intentions lists in this scheme as a recovery mechanism avoids the occurrence of information flow or obsolescence – – the reason for commit-dependency and abort-dependency formation – – between active transactions since the modifications of an object by an operation are not effected until the operation commits. For example, with intentions lists, if a Pop operation follows a Push operation invoked by different transactions on a QStack, information flows from Push to Pop only when the Pop commits. At the time of commitment, a transaction is validated to determine if its commitment invalidates the changes made by any committed transaction in case of backward validation, or the effects of any in-progress (active) transaction in case of forward validation.

Recoverability is another criterion which is used to define conflicts among operations[3, 2]. An operation o_1 is *recoverable* relative to another operation o_2, if o_2 returns the same value whether or not o_1 is executed immediately before o_2. Transactions invoking o_1 and o_2 are required to commit in the order of invocation of these two operations. Since recoverability forces a (dynamically determined) order of commitment for active transactions, in a sense it is stronger than serial dependency which postpones the commit order till the time of commitment of active transactions. Recoverability, like commutativity, allows implementations that avoid cascading aborts while also avoiding the delay in the processing of many non-commutative operations. It assumes a flexible recovery technique for handling the abortion of operations.

In both serial dependency and recoverability, aspects of input/output data semantics relating to input and return values, and operation semantics are used. Both these definitions are weaker notions than commutativity which requires equivalence of states. In fact serial dependency and recoverability can be shown to be equivalent semantic notions in the sense

Pop(): e/nok deletes an element e from the back of the QStack. It returns e if QStack is not empty, *nok* (denoting *empty* or *bottom*) otherwise.

Top(): e/nok returns e, the element at the back of QStack, if QStack is not empty, *nok* (denoting *empty* or *bottom*) otherwise.

Size(): n returns the number of elements n in the QStack.

Replace(e1,e2): ok replaces all $e1$ elements (values) in QStack with $e2$. It always returns *ok*.

XTop(): ok/nok exchanges the first two elements in the back of the QStack. It returns *ok* if two elements exist, otherwise *nok*.

We refer to the "status", such as *ok* or *nok*, returned by an operation as the *outcome* of the operation. Other values returned are referred to as its *result*. It is assumed that an operation always produces a return-value, that is, it has an outcome or a result or both.

2.1 Effects of Operations on Each Other

Operations defined on an object are considered as functions from one object state to another object state. The result of an operation on an object depends on the current state of the object. For a given state s of an object, we use $return(s,p)$ to denote the return value, i.e., result or outcome, produced by operation p, and $state(s,p)$ to denote the state produced after the execution of p.

Here we ask the question: What are the possible interactions that can occur between two concurrent operations on a given object, and what are the effects of these interactions on the relationship between these two operations? This relation can cause dependencies to develop between the transactions invoking the two operations, thus affecting their commit or abort. The relationship between an operation and another relative to state s depends on whether it is an observer of s or a modifier of s, or both. Two operations interact only if at least one of them is a modifier that changes the state of the object.

> **Definition 1:** An operation o is an *observer* in a state s if $state(s,o) = s$.
>
> **Definition 2:** An operation o is a *modifier* in a state s if $state(s,o) \neq s \wedge \forall s', s' \neq s, return(s',o) = return(s,o)$.
>
> **Definition 3:** An operation o is a *modifier-observer* in state s if $state(s,o) \neq s \wedge \exists s', s' \neq s, return(s',o) \neq return(s,o)$.

These definitions classify the operations as observer (O_s), modifiers (M_s) and modifier-observers (MO_s) with respect to a particular state s. Clearly an operation could be a modifier in one state and an observer in another. For example, operation Push is a modifier-observer if successful and just an observer if it is not.

Interactions between operations of different classes can cause dependencies of different types between the invoking transactions. Given the above definitions, consider the cases where an operation q follows an operation p. If p is a modifier and q is an observer, or p is modifier and q is a modifier-observer, or p is a modifier-observer and q is an observer, or both p and q are modifier-observers, operation q observes the effects of p. In these cases, to guarantee failure atomicity, the transaction invoking q has to abort if for some reason the first transaction aborts, since the information used by q would no longer be valid. The second transaction can commit only if the first transaction commits. Hence, in this case, the second transaction is said to be *abort-dependent (AD)* on the first transaction.

If p is an observer and q is a modifier, or p is an observer and q is a modifier-observer, or p is a modifier-observer and q is a modifier, or both p and q are modifiers, the outcome and result of q are not affected by the effects of p. In these cases, to ensure serializability, if both transactions commit, the first should commit before the second. i.e., the second transaction can commit, and hence, it can effect its changes, only after the first transaction commits or aborts[1]. In this case, the second transaction is said to be *commit-dependent (CD)* on the first transaction.

Thus far, we have classified operations in a given state s. Let us consider *state-independent* classification of operations. To motivate this classification observe that an abort-dependency is stronger that a commit-dependency in the sense that abort-dependency can prevent a transaction from committing and thus force it to abort, whereas commit-dependency can neither prevent a transaction from eventually committing nor force it to abort. (Note that abort-dependency implies commit-dependency.) Because of this, suppose an operation o is a modifier-observer in a state s, there is potential for another operation to form an abort-dependency on o; also, suppose o is a modifier in another state s' in which there is potential for another operation to form a commit-dependency on o; o should be classified as a modifier-observer with respect to all states because abort-dependency is stronger than commit-dependency. Here is a state-indenpendent classification of operations:

> **Definition 4:** An operation o is a modifier-observer (MO), if $\exists s$ in which o is a modifier-observer.
>
> **Definition 5:** An operation o is a modifier (M) if $\nexists s$ in which o is a modifier-observer, and $\exists s$ in which o is a modifier.
>
> **Definition 6:** An operation o is an observer (O) if $\nexists s$ in which o is a modifier-observer, and $\nexists s$ in which o is a modifier.

Here is a state-independent classification of the operations of the QStack:

Operation	Type	Operation	Type
Pop	MO	Deq	MO
Push	MO	Size	O
Top	O	Replace	M
XTop	MO		

Table 1

[1] This means that, in the event that p is aborted, p's changes have to be undone and possibly q's, and the changes of q must be reapplied.

that they allow the same set of valid histories given a particular recovery mechanism. We have proven elsewhere that both these schemes have the same set of valid histories, and we have shown how a serial dependency based compatibility table translates into a recoverability table and vise-versa. The difference between these two semantic notions is in the assumption of the underlying recovery mechanism.

Compatibility of operations based on the formation of significant and insignificant dependencies between concurrent operations is described in [13]. For example, two concurrent read operations form an insignificant dependency and hence can be allowed to execute concurrently. The classification of dependencies as significant or insignificant is not explicitly uniform across types. Here a combination of operation, input/output data, and object organization semantics is exploited.

Whereas each of these proposed semantics-based concurrency control schemes has attempted to exploit different aspects of objects and their operations, it is not clear if (between them) that have exhausted all possibilities. It will be useful to know – given a particular object – what the potential for concurrency is while executing operations on the object. This is the issue addressed in the next section. Specifically, we develop a model for representing objects that brings the concurrency properties of an object and its operations to the forefront.

4 The Object Model

The popular notion of an object is that it hides or encapsulates implementation details, and presents only the logical or abstract view of the objects, with a predefined set of methods or operations that are used to access the object. Even while staying within this view, one can exploit another abstract object characteristic, namely, the notion of ordering among elements[2]. Specifically, an object can be thought of as containing a set of subobjects or components ordered in a specific way. This potentially rich source of semantic information can be used to extract more concurrency.

In this section, we develop a characterization of objects that helps identify the semantic information inherent to an object. First we propose use of an object graph to represent objects and their components. Using this graph, the *locality* of effects of an operation is described. Locality of operations forms the basis for deriving their concurrency properties.

4.1 Object Graph

Objects are instances of abstract data types whose state can be observed and manipulated by a set of operations defined on the object. The state can be viewed as an ordered set of component objects. In this recursive view, the primitive object has a simple data value. Hence this view captures both the notions of *simple* and *complex* objects, i.e., objects composed of other objects.

Definition 7: An object *ob* is a 3-tuple (S, R, O) where S is a set of objects or simple data values, R is a set of ordering rules, and O is a set of operations defined on *ob*.

[2]Note that not all objects may contain components that are ordered. However, as we will see, where available, such ordering information can be exploited to improve concurrency.

The object graph represents the logical organization (abstract structure) of an object. This graph encodes the fact that an object consists of component objects, where the ordering among components, represented by R, is made explicit by encoding it as edges in the graph. The ordering edge emanating from a component indicates the next component that can be accessed following access to this component. A component may be an object with or without further components. This representation, which is an extension of the graph model used in [1], can be constructed just from the abstract specification of an object and the operations and does not subsume any implementation detail. In particular, we are dealing with the abstract operations for which it is assumed that the implementation does not impose any constraints on extracting the concurrency inherent in an object.

Thus, in what follows, it is assumed that if the semantics of two operations on an object allow the operations to execute concurrently, the lower-level implementation of the object will allow the exploitation of such concurrency. In case an object has components which are themselves objects, then concurrent access to that object (perhaps from operations invoked on the parent object) are controlled by the component object. Such multilevel concurrency control issues pertaining to complex objects [9] are studied in [11, 10, 2].

Definition 8: Let $G_{ob}(\{v_{ob} \cup V_{ob}\}, \{E_{com} \cup E_{ord}\})$ be the *object graph* for object *ob* where:

- v_{ob} is the root of the object graph,
- V_{ob} is a set of vertices, representing the components of *ob*,
- E_{com} is a set of *composed-of* edges from v_{ob} to every vertex in V_{ob}, representing the composition of *ob*, and
- E_{ord} is the set of *ordering* edges connecting vertices in V_{ob}, representing the ordering of the components of *ob*.

Definition 9: The subgraph $G'_{ob}(\{v_{ob} \cup V_{ob}\}, E_{com})$ of G_{ob} is called the *composition* graph of *ob*, and subgraph $G''_{ob}(V_{ob}, E_{ord})$ is called the *ordering* graph of *ob*.

Definition 10: The *content* of a vertex is defined recursively as follows:

- If the vertex denotes an object without components, then the content of the vertex is the content of the object.
- If the object has components, the content of the vertex is denoted by the composed of edges and content of the vertices in the subtree rooted at the vertex.

For example, in Figure 1, object A is composed-of primitive objects B and C and component object D which is composed-of primitive objects E and F. Since D itself is an object, A is a complex object. AB, AC, AD, DE and DF are composed-of edges (the solid arrows). Thus, the composition graph of A is $G'(\{A, B, C, D\}, \{AB, AC, AD\})$. The ordering edges of A are BC and CD (the dotted arrows). Thus,

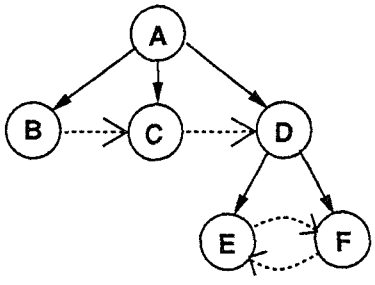

Figure 1: An object graph

the ordering graph of A is $G''(\{B,C,D\},\{BC,CD\})$. Note that DE and DF are composed-of edges and EF and FE are ordering edges of D and not of A.

The composition graph of an object together with the composition graphs of component objects has a hierarchical structure. At any level of the hierarchy, the ordering graph of the object at that level may contain cycles. Since ordering is meaningful only between components of an object, ordering edges are restricted to lie at a single level, i.e., they do not connect vertices at different levels.

4.2 Operations and Locality

An operation can possibly do one or more of the following:

1. change the contents of vertices (for component objects, by invoking operations on them),
2. insert or delete vertices and their related (composed-of and ordering) edges,
3. change the structure by changing the ordering edges,
4. observe the contents of vertices, or
5. observe the structure or presence of vertices.

Out of these, item 5 might need some further explanation. The following example should clarify it. Consider the operation Size on the QStack object. The number of elements on a QStack that Size returns equals the number of vertices present in the structure of the QStack (see Figure 2). Thus, Size observes the structure and counts the vertices present, i.e., it observes the presence of the vertices.

To characterize the specific parts of the object graph affected by or used by an operation, we define the locality of an operation. Note that an operation on a complex object may result in the invocation of a set of operations on the component objects (see item 1 above).

> **Definition 11:** The *locality* L_o of an operation o is a set of vertices inserted or deleted by o, vertices whose existence has been observed by o, vertices whose content has been changed or observed by o and vertices to/from which ordering edges have been changed or observed.

Since operations can affect the structure of an object and the contents of its components, the locality of an operation o can be split into two sets L_o^s and L_o^c ($L_o = L_o^s \cup L_o^c$), not necessarily disjoint, denoting the *structure* and *content* locality of o respectively.

> **Definition 12:** The *structure locality* L_o^s of an operation o is a subset of the locality of the operation L_o ($L_o^s \subset L_o$), containing the vertices in L_o that are inserted or deleted, vertices to/from which ordering edges are changed or observed, and vertices whose presence is observed.

> **Definition 13:** The *content locality* L_o^c of an operation o is a subset of the locality of the operation L_o ($L_o^c \subset L_o$), containing the vertices in L_o that are inserted, deleted, or whose content is changed or observed.

Informally, whereas structure locality of an operation considers all the vertices whose *existence* or *ordering*, may have been noted by the operation, content locality considers all the vertices whose *contents* may have been observed or affected by the operation.

By considering insert, delete, and change operations as *modifiers*, structure as well as content localities of an operation can be further distinguished into *structure-observation*, *structure-modification*, *content-observation*, and *content-modification* localities.

> **Definition 14:** The *structure-observation* locality L_o^{so} of an operation o is a subset of the structure locality of the operation L_o^s ($L_o^{so} \subset L_o^s$), containing the vertices in L_o^s whose presence is observed, and ordering edges to/from which are observed.

> **Definition 15:** The *structure-modification* locality L_o^{sm} of an operation o is a subset of the structure locality of the operation L_o^s ($L_o^{sm} \subset L_o^s$), containing the vertices in L_o^s that are inserted or deleted, or to/from which ordering edges are changed.

> **Definition 16:** The *content-observation* locality L_o^{co} of an operation o is a subset of the content locality of the operation L_o^c ($L_o^{co} \subset L_o^c$), containing the vertices in L_o^c whose content is observed.

> **Definition 17:** The *content-modification* locality L_o^{cm} of an operation o is a subset of the content locality of the operation L_o^c ($L_o^{cm} \subset L_o^c$), containing the vertices in L_o^c that are modified, i.e., inserted or deleted, or whose content is changed.

The following examples should clarify these terms. The Replace operation on a QStack modifies the content but not the structure of the QStack. The Top operation observes both the content as well as the structure of the QStack. A successful XTop modifies the structure but not the content of the QStack. That is because XTop which affects the top two elements of the QStack does not modify the content of the elements or the composed of edges. It only reorders the ordering edges.

It is possible for an operation to have more than one characterization. For instance, XTop, as specified, is characterized by a non-empty structure-modification locality and empty content-modification locality. If XTop were to swap the contents the top two elements, then XTop would modify the content of the QStack but not the structure. In this

case, XTop would be characterized by a non-empty content-modification locality and empty structure-modification locality. In such cases, all characterizations need to be considered in order to determine the one characterization that allows the most concurrency.

Definition 18: V_{ob}^{simple} for an object *ob* is the set of all the vertices with simple data values in the hierarchical graph constructed by the composition graph of the object $G_{ob}^{'}$ together with the composition graphs $G_{ob_i}^{'}$ of its component objects ob_i ($ob_i \in V_{ob}$). Thus, $V_{ob}^{simple} = V_p \cup (\cup_i V_{ob_i}^{simple})$, where: $V_p \subset V_{ob}$, represent the primitive components of *ob*.

Definition 19: An operation *o* on an object *ob* is said to be a *global* operation if $L_o \supset V_{ob}^{simple}$. An operation which is not global is said to be *non-global*.

That is, the locality L_o of a global operation *o* defined on *ob always* contains all the vertices with simple data values in the object graph of *ob*.

Global operations can be classified as *global-modifiers* and *global-observers*, or can be further refined as *global-structure-modifiers*, *global-structure-observers*, *global-content-observers* and *global-content-modifiers* according to the locality type. For instance, if for an operation *o*, $L_o^{co} = V_{ob}^{simple}$ then *o* is said to be a global-content-observer. Replace is an example of such an operation.

4.3 Relation of Locality to Dependencies

In the most general case, two operations defined on an object conflict if the intersection of the localities of the two operations is not empty. We focus first on the state-independent classification of operations and then we show how return-value dependency and state-dependency semantics can be factor in.

Finding the actual locality of an operation may require the execution of the operation. However, in most cases the locality of a non-global operation can be specified by a predicate. Thus, whether the intersection of two localities is empty or not can be determined by using the predicates characterizing the localities without actually finding the vertices or edges involved. (However, note that, in general, determining if the sets identified by two arbitrary predicates intersect is undecidable.) If it is not possible to specify such predicates, the locality of an operation can be determined only after the operation completes.

Typically, the input parameters to an operation determine the locality of the operation. In addition, the ordering among component objects can be very effectively used in constructing predicates for specifying the localities of the operations. To this end, the notion of the set of *references* used by each operation on an object is introduced. This set is a subset of the composed-of edges E_{com} emanating from the root vertex of the object graph and is generally maintained as part of the object state.

For example, a QStack maintains two references, one is the *back pointer* or *stack pointer* (denoted by *b* in Figure 2) that points to the end of the QStack and is used by Enq, Push, Pop, and Top operations, and the other is the *front pointer* (denoted by *f* in Figure 2) that points to the front of the queue and is used by the Deq operation. The stack pointer, for instance, is the composed-of edge corresponding to the last element on the QStack. The ordering edges define which composed-of edge should become the stack pointer when a Pop operation is invoked (since the composed-of edge representing the current stack pointer is deleted when Pop executes).

Definition 20: Let r_{op} be the set of references of operation *op* defined on object *ob* represented by the object graph $G_{ob}(\{v_{ob} \cup V_{ob}\}, \{E_{com} \cup E_{ord}\})$. The set of references r_{op} is a subset of the composed-of edges E_{com} ($r_{op} \subset E_{com}$) and is defined with respect to *ob*'s ordering rules R (see definition 7).

These references can either be (indirectly) provided by the agent invoking the operation (*explicit referencing*), or by the object state itself that maintains a set of references to be used by the operations (*implicit referencing*). The two references maintained by a QStack are implicit. An example of explicit referencing occurs in the $search(x)$ operation on a relation. Here x is the argument that can be used to determine the reference to the record being searched.

References can be deleted for example when a QStack becomes empty. A reference can also be modified. Modification can be done without necessarily deleting the corresponding composed-of edge by selecting a different composed-of edge as the new reference. For example, a Push operation on a QStack modifies the stack pointer by selecting the newly added composed-of edge to be the new stack pointer but without deleting the composed-of edge representing the current stack pointer.

Definition 21: The input and output parameters of an abstract operation on an object can be said to contain three components: reference (r), input-data (i), return-value(o).

Assertion 1: Transactions invoking two operations x and y defined on the object *ob* do not form dependencies if localities of x and y satisfy the following conditions:

$$L_x^{cm} \cap L_y^{co} = L_x^{co} \cap L_y^{cm} = L_x^{cm} \cap L_y^{cm} = L_x^{so} \cap L_y^{sm} = L_x^{sm} \cap L_y^{so} = L_x^{sm} \cap L_y^{sm} = \phi$$

□

This implies that operations restricted to the structure of an object do not form dependencies with operations restricted to the content of the object.

The intersection of different combinations of locality types, if it is not empty, may result in either a commit or an abort dependency. In the following table, if the intersection of the given sets is empty then no dependency (ND) is developed between the corresponding operations x and y, otherwise an abort-dependency (AD) or a commit-dependency (CD) is developed. In all the compatibility tables, for better readability, an ND is indicated by a blank entry. Here x is in execution and y attempts to execute concurrently with x.

	L_x^{so}	L_x^{co}	L_x^{sm}	L_x^{cm}
L_y^{so}			AD	
L_y^{co}				AD
L_y^{sm}	CD		CD	
L_y^{cm}		CD		CD

Table 2

Assertion 2: Given two operations x and y defined on the object ob, x and y *commute* iff $\forall i \in \{c,s\}, \forall k,l \in \{o,m\}, \neg(k = o \wedge l = o), L_y^{ik} \cap L_x^{il} = \phi$. □

Assertion 3: Given two operations x and y defined on the object ob, y is *recoverable* relative to x iff $\forall i \in \{c,s\}, \forall k,l \in \{o,m\}, \neg(k = o \wedge l = o)$, (a) $L_y^{ik} \cap L_x^{il} = \phi$ or (b) $L_y^{ik} \cap L_x^{il} \neq \phi$ and the corresponding entry in Table 2 is a ND or a CD. □

4.4 Putting it all together

We are now in a position to determine the *compatibility table* associated with a given object.

In the traditional framework, a compatibility table is a simple a binary relation with a *yes* entry for (o_i, o_j) indicating that the operations o_i and o_j are compatible, i.e., do *not conflict*, or a *no* entry indicating that the two operations are incompatible, i.e., conflict. In our terminology, an entry could contain *no-dependency (ND)*, *abort-dependency (AD)*, or *commit-dependency (CD)*. That is, in our scheme a standard *yes* entry translates to a ND, whereas a standard *no* entry is refined to either AD or CD. Note (see Section 2 for the definitions of dependencies) that an AD entry is more restrictive (*stronger*) than a CD entry, and a CD entry is more restrictive than a ND entry (AD>CD>ND). The general rule to determine an entry in a table follows from the discussion of the effects of an operation on another (see Section 2.1).

In this scheme, the compatibility table is developed through stepwise refinement of its entries. Each step uses more semantic information to produce a compatibility table that offers more potential for concurrency among operations.

We begin with the case where no semantic information is used about the object and its operations, i.e., corresponds to all operations being modifier-observers (MO). This produces a single entry compatibility table containing AD (X is the operation in execution and Y is the invoked operation).

	X
Y	AD

Table 3

Based on whether an operation is an observer (O), a modifier (M), or modifier-observer (MO), this single entry table can be replaced by the following table:

	O	M	MO
O		AD	AD
M	CD	CD	CD
MO	CD	AD	AD

Table 4

These entries capture the eight types of potentially conflicting interactions between two operations seen in Section 2.1. This is exactly the semantics that is captured by *recoverability* [and *serial dependency*].

By making use of the order among dependencies (AD>CD>ND), the entries associated with a modifier-observer can be considered as a function that returns the stronger dependency between the corresponding modifier and observer entries. For example, the entry (O,MO) = *stronger*((O,M),(O,O)) = *stronger*(AD,ND) = AD. Since the MO entries can be easily generated in this way, we need to further consider tables with only the O and M entries:

	O	M
O		AD
M	CD	CD

Table 5

Note that the above "weakening" was accomplished by using a combination of semantics of both operations.

Object organization semantics, i.e., the composition and structural (ordering) semantics of an object, is used to further refine the AD and CD entries. An observer can either be a *content observer (CO)* or a *structure observer (SO)* or both *(CSO)*. Similarly, a modifier can be classified as *content modifier (CM)*, *structure modifier (SM)*, or both *(CSM)*. It is possible to eliminate the entries associated with CSM by making use of the *stronger* function as explained above. Operations restricted to the structure of an object can execute concurrently with operations restricted to the content of the object. For example, the operation Replace defined on a QStack is a CM operation on the QStack and a successful XTop is a SM operation on the QStack, and hence, Replace and successful XTop operations commute. The Top operation on the other hand is both SO and CO since it observes both the ordering and the content of the first node. By this refinement, based on the entries in Table 2, the following three tables are obtained corresponding to the entries (O,M), (M,M), and (M,O):

(O,M)	SM	CM	CSM
SO	AD		AD
CO		AD	AD
CSO	AD	AD	AD

Table 6

(M,M)	SM	CM	CSM
SM	CD		CD
CM		CD	CD
CSM	CD	CD	CD

Table 7

(M,O)	SO	CO	CSO
SM	CD		CD
CM		CD	CD
CSM	CD	CD	CD

Table 8

The refinement thus far was based on a state-independent characterization of operations. The input/output semantics of operations and the locality of non-global operations can be exploited to further weaken the remaining AD and CD entries. Potentially, these are very rich sources of semantic information that can be effectively used to further enhance concurrency. This is illustrated in Section 5.

The operations can be classified as either global (G) or non-global (L) and only the non-global operations need to be refined further: The single dependency in an entry is replaced with a set of mutually-consistent (dependency/condition) pairs where each condition is dependent on the predicate that describes the locality of the operation (see Section 4.3). Different conditions test the emptiness of the intersection of different types of localities of two operations and may result in different dependencies. By mutually-consistent (dependency, condition) pairs, we mean that if the conditions associated with two pairs involve the same type of localities where the condition of the first pair exploits more semantics than the one of the second pair, the dependency specified in the first pair must be weaker than the one specified in the second pair. Thus, for a given entry, the dependency chosen from the set of (dependency,condition) pairs is the least restrictive (weakest) dependency among the dependencies whose associated conditions hold.

Let us consider a Push operation followed by a Deq operation. The entry (Deq,Push) of the compatibility table of QStack contains the pair $(CD, Push_{out} = nok)$, since an unsuccessful Push (returning *nok*) is only an observer, and hence, Deq has a CD with this Push. As we will see in the next section, the entry (Deq,Push) also contains the pair $(ND, f \neq b)$ (Recall that f and b stand for the current front and back pointer of the QStack), denoting that the intersection between the localities of operations Push and Deq is empty. That is, for the state in which $f \neq b$, Push and Deq commute. In the event of an unsuccessful Push, both conditions become true, and hence, ND should be chosen, being the weaker of ND and CD.

Based on the above discussion, a methodology presents itself that helps to generate the conflict resolution table for an object methodically. This table is a $n \times n$ table, where n is the number of operations defined on the object, with AD, CD, or ND entries which could be either conditional or unconditional. Conditional entries are those that are based on dynamic information such as the locality of a non-global operation. The objective is to obtain an optimal table that contains the minimal of AD or CD entries, with CD preferred over AD, and weaker conditional entries preferred over stronger unconditional entries.

In the next section, we first discuss this methodology and then apply it to the QStack example.

5 The Methodology

In the *stage 1*, in order to identify all the necessary information, the object graph G_{ob} of the object is constructed and the references are identified. Then the behavior of each operation defined on the object is expressed in terms of operations on G_{ob}.

In the *stage 2*, using information gathered in the first stage, for each operation, answers to the following questions are sought.

D1: Is it an *observer, modifier* or *modifier-observer*?

D2: Does it observe/modify *content, structure*, or both?

D3: Does it have an *outcome*, or *result*, or both? Does it have input parameters?

D4: Is its *locality global* or not?

D5: Does it employ *explicit* or *implicit* referencing, if implicit, which are the references used?

$D1$ and $D2$ involve state-independent semantics, $D3$ is related to input/output semantics, and $D4$ and $D5$ are state dependent semantics.

In the *stage 3*, an initial compatibility table T_{ob} of the object is derived from Tables 5, 6, 7, and 8, collectively referred to as *template tables*, using the first two dimensions, namely $D1$ and $D2$, of the characterization of the operations produced in stage 2. For each pair of operations (o_1, o_2) where each operation is either a modifier, an observer, or a modifier-observer, potentially every one of these template tables can specify a dependency. Specifically, for such a pair of operations the corresponding entry in T_{ob} is determined from the following tables:

- Table 5, if both operations can be characterized in terms of $D1$.
- Tables 6, 7, and 8, if both operations can be characterized in terms of $D2$.

The final dependency for the pair of operations (o_1, o_2) is taken to be the least restrictive dependency of the dependencies specified by the appropriate template tables in each dimension.

Since modifier-observer operations are considered to be a composition of modifier and observer operations, two dependencies will result, one for each component, along each dimension. The single dependency for modifier-observer along each dimension is chosen, as explained above, to be the more restrictive of the resulting two dependencies.

In the *stage 4*, the D3 dimension of the characterization of operations is used to refine entries. This is achieved by replacing the existing AD or CD dependency with a set of (dependency,condition) pairs where the conditions are based on the outcome and on input parameters, and the dependency in at least one of the pairs is less restrictive than the existing one.

In the *stage 5*, the final stage, using the dimensions D4 and D5 of the characterization of the operations, all non-global operations are identified, and their locality predicate is constructed in terms of their input parameters and/or their references. For every pair (o_1, o_2) of non-global operations their corresponding entry in the initial compatibility table is added or replaced with a set of (dependency,condition) pairs, where conditions are expressed in terms of the constructed predicates.

We now generate the compatibility table for the QStack as an example. For this purpose, we focus on the following operations defined on QStack: Push, Pop, Deq, Size, and Top.

In *stage 1*, a graph representation for QStack is constructed, as shown in the Figure 2. The ordering edges (dotted arrows) point towards the front of the QStack. QStack maintains two implicit references f (front pointer) and b (back pointer) that are the composed-of edges pointing to the first and last element of QStack respectively. These references are used by the operations to access the elements.

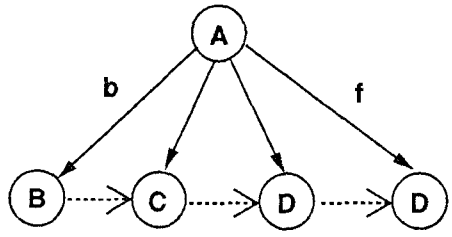

Figure 2: Object graph for QStack

In *stage 2*, all the operations defined on QStack are characterized along the dimensions D1-D5 as stated above. The characterization of each operation is summarized in Table 9.

Op.	obs/ mod	Cont/ Str	return-value	Locality	Reference
Pop	MO	CS	*result/nok*	L	*f*
Push	MO	CS	*ok/nok*	L	*f*
Deq	MO	CS	*result/nok*	L	*b*
Size	O	S	*result*	G	
Top	O	CS	*result/nok*	L	*f*

Table 9

The reason that Size is not associated with a reference is that Size counts the composed-of edges in the object graph of QStack and counting of composed-of edges does not require any specific order. Counting could start from any composed-of edge, and hence, need not refer to *f* or *b*.

In *stage 3*, the entries for each pair of operations is determined by consulting the tables presented in section 4.4. For example, the entry for the operation pair (Deq, Push)[3] is determined as follows:

1. Based on D1, Push and Deq are of type MO and MO respectively, and hence the stronger of the entries we get from Table 5 for (M,M), (M,O) and (O,M) is a AD.

2. According to D2, both these operations are of type CS. The stronger of the entries from Table 6 and 7 for (CSM,CSM), (CSM, CSO) and (CSO,CSM) is again a AD.

The compatibility table for QStack obtained as a result of *stage 3* is as follows:

(o_1, o_2)	*Push*	*Pop*	*Deq*	*Top*	*Size*
Push	AD	AD	AD	CD	CD
Pop	AD	AD	AD	CD	CD
Deq	AD	AD	AD	CD	CD
Top	AD	AD	AD		ND
Size	AD	AD	AD		ND

Table 10

In *stage 4*, based on D3, the outcome of the operations are used to refine the entries. Focusing again on the (Deq, Push) entry, although both the operations have outcomes, only the outcome of the Push operation helps in refining the existing dependency by replacing it with the set of (dependency,condition) pairs: $\{(AD, Push_{out} = ok), (CD, Push_{out} = nok)\}$. This is because, when the outcome is *nok*, Push acts as an observer and not as a modifier-observer. In a similar manner Table 12 can be constructed for (Push,Push).

	Push
Deq	(CD,$\text{Push}_{out} = nok$) (AD,$\text{Push}_{out} = ok$)

Table 11

	Push^x
Push^y	(ND,Push^x_{out}=$\text{Push}^y_{out} = nok$) (CD,$\text{Push}^x_{out} = nok \wedge \text{Push}^y_{out} = ok$) (CD,$\text{Push}^x_{out} = \text{Push}^y_{out} = ok$) (AD,$\text{Push}^x_{out} = ok \wedge \text{Push}^y_{out} = nok$)

Table 12

Now we can consider further refinements based on input parameters. For example, if two Push operations attempt to push the same element *e*, they commute.

	Push^x
Push^y	(ND,Push^x_{out}=$\text{Push}^y_{out} = nok$) (CD,$\text{Push}^x_{out} = nok \wedge \text{Push}^y_{out} = ok$) (CD,$\text{Push}^x_{out} = \text{Push}^y_{out} = ok$) (AD,$\text{Push}^x_{out} = ok \wedge \text{Push}^y_{out} = nok$) (ND,$\text{Push}^x_{in} = \text{Push}^y_{in} = e$)

Table 13

In *stage 5*, the entries corresponding to non-global operation pairs are refined further. Considering the example (Deq, Push) pair, both Push and Deq are non-global, based on D4. Therefore, as we show now, some of the corresponding (dependency,condition) pairs of (Deq, Push) entry can be replaced with pairs involving weaker dependencies and conditions expressed in terms of locality predicates.

Based on D5, we can note that both Push and Deq employ implicit referencing and use the references *b* and *f* respectively. This means that the intersection of their localities could be empty, in which case there will be no dependency (ND). The intersection between the localities of Push and Deq can be determined by a predicate constructed from the references *f* and *b* that tests whether before the operations are executed *f* and *b* refer to the same composed-of edge, i.e., refer to the same component object. Hence the (dependency,condition) pair having the AD in (Deq, Push) is replaced, and the (Deq, Push) entry becomes:

	Push
Deq	(CD,$\text{Push}_{out} = nok$) (AD,$f = b$) (ND,$f \neq b$)

Table 14

The entries for the remaining pairs can be refined by following the same procedure for stages 4 and 5.

To summarize the methodology just used, given an object, the specific operations defined on the object are expressed in terms of operations on the graph representation of the object.

[3]Recall that (Deq, Push) entry corresponds to the situation that a Deq operation follows a Push operation on the QStack.

The compatibility of each pair of operations is determined by using the produced graph characterization of the operations and the template tables. Subsequently, each entry may be refined by considering input/output semantics and by defining conditions in terms of locality predicates.

6 Conclusion

Whereas a number of semantics-based concurrency control schemes for object-oriented systems have been proposed in the literature, each scheme has approached the issue from fairly narrow considerations. In this paper, we have approached the problem from first principles in an effort to discover the underpinnings of, and hence classify, existing schemes while giving a unified view to the nature of semantics inherent in objects.

In this regard, we have classified the semantic information available within an object in order to identify the specific combinations that can possibly yield enhanced concurrency. To formalize this classification, we have proposed an object model and its graph representation that can be derived from abstract specification of an object. We have shown how the model can be effectively used to identify the available semantic information about an object.

We have proposed a scheme that methodically exploits the available semantic information. This shows how various semantic notions applicable to concurrency control can be effectively combined to achieve improved concurrency. In this process, we have identified and exploited a new source of semantic information, namely, the ordering among component objects, to further enhance concurrency.

We have also classified the semantic information into static and dynamic information, depending on when it is available, to facilitate easy design of compatibility tables. To determine dynamic information such as the locality of an operation, we have provided a framework or ground rules within the proposed object model, that can be effectively used to identify further possibilities of improved concurrency. Lastly, and perhaps of the most practical interest, we have presented a methodology for deriving compatibility tables for operations on objects. Note that no assumption regarding the underlying optimistic or pessimistic concurrency control mechanism as well as the recovery mechanism has been made in deriving the compatibility tables. When specific mechanisms are considered, tables can be refined further.

In this paper, we did not make use of *usage semantics*, but this semantics is extensively utilized in various extended transaction models that relax the requirements of serializability and failure atomicity to achieve more concurrency. Just as the present paper has attempted to unify object semantics, the ACTA framework introduced in [5] provides a unifying framework for all these transaction models. The notion of dependencies among transactions serves as the thread common to both these efforts. Hence, by using the results of this paper in conjunction with the ACTA model, the semantic information obtained from the relaxed correctness requirements of an application can be used to further enhance concurrency.

References

[1] Badrinath, B. and Ramamritham, K. Synchronizing Transactions on Objects. *IEEE Transactions on Computers*, 37(5):541–547, May 1988.

[2] Badrinath, B. and Ramamritham, K. Performance Evaluation of Semantics-based Multilevel Concurrency Control Protocols. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 163–172, Atlantic City, NJ, May 1990.

[3] Badrinath, B. and Ramamritham, K. Semantics-based concurrency control: Beyond Commutativity. *(to appear in) ACM Transactions on Database Systems*, 1991.

[4] Bernstein, P. A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[5] Chrysanthis, P. K. and Ramamritham, K. ACTA: A Framework for Specifying and Reasoning about Transaction Structure and Behavior. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 194–203, Atlantic City, NJ, May 1990.

[6] Eswaran, K., Gray, J., Lorie, R., and Traiger, I. The notion of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[7] Gray, J. The transaction concept: Virtues and limitations. In *Proceedings of the 7th VLDB Conference*, pages 144–154, September 1981.

[8] Herlihy, M. P. and Weihl, W. Hybrid concurrency control for abstract data types. In *Proceedings of the 7th ACM symposium on Principles of Database Systems*, pages 201–210, March 1988.

[9] Kim, W., Banerjee, J., Chou, H.-T., Garza, J., and Woelk, D. Composite object support in an Object-Oriented database system. In *Proceedings of Object-Oriented Programming Systems, Languages, and Applications*, pages 118–125, Orlando, Florida, October 1987.

[10] Martin, B. E. Modeling concurrent activities with nested objects. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 432–439, Berlin, Germany, September 1987.

[11] Moss, J. E. B., Griffeth, N., and Graham, M. Abstraction in recovery management. In *Proceedings of the ACM SIGMOD international conference on management of data*, pages 72–83, May 1986.

[12] Roesler, M. and Burkhard, W. Concurrency Control Scheme for Shared Objects: A Peephole based on Semantics. In *Proceedings of 7th International Conference on Distributed Computing Systems*, pages 224–231, September 1987.

[13] Schwarz, P. M. and Spector, A. Z. Synchronizing shared abstract data types. *ACM Transactions on Computer Systems*, 2(3):223–250, August 1984.

[14] Stonebraker, M. *(Ed.). Readings in Database Systems.* Morgan Kaufmann, 1988.

[15] Weihl, W. Commutativity-Based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.