TR 05-004

Semantics Based Commutativity Analysis of Object Methods

John Eberhard and Anand Tripathi

March 04, 2005

Semantics Based Commutativity Analysis of Object Methods

John Eberhard and Anand Tripathi
Department of Computer Science and Engineering
University of Minnesota
{eberhard, tripathi}@cs.umn.edu

Abstract

Traditional analysis and usage of operation commutativity relies on pairwise commutativity relationships. In contrast, this paper presents *method group* commutativity, which specifies the conditions under which operations in a *method group* will commute. Method group commutativity can be practically applied to efficiently support distributed object caching and concurrency control. A formal definition of commutativity, in terms of Hoare logic expressions, guides the development of a methodology to create a *method commutativity specification* from an object's semantic specification. This methodology uses the PVS theorem prover for analysis and validation of commutativity properties. The use of formal methods, together with suitable tools, provides a more complete understanding of method commutativity relationships as compared to existing approaches. Our approach also provides a unified representation of both *forward* and *backward* commutativity properties. The methodology is also expanded to express weakened semantics in the method commutativity specification to enable greater commutativity.

I. Introduction

Synchronization of concurrent operations on a shared object is required to preserve its consistency. The concurrency control techniques in object-based systems have ranged from locking protocols based on read/write sets of operations to those that exploit semantics of the operations. The need for object-based concurrency arises in both centralized as well as distributed systems. For distributed applications in wide-area networks, caching and replication of shared objects is generally needed to ensure acceptable performance. However, concurrent updates on cached copies in such environments require efficient synchronization techniques. The focus of this paper is on formal analysis of semantics of object methods, particularly in regard to their commutativity properties, to support greater concurrency in the execution of object methods. Efficient concurrency control and synchronization techniques are necessary to effectively support replication and caching, particularly for wide-area networks, mobile computing environments, and real-time systems.

When considering the semantics of operations, two operations do not conflict if they commute. Intuitively, two operations commute if the computational effects of executing the operations are the same, independent of the order in which the operations are executed. Previous work on commutativity has typically determined the commutative relationship between pairs of operations. We expand on this work by determining when a group of operations will commute.

Our work uses a semantic specification, given in first-order logic, of object methods in an object-based system to determine the commutativity properties of those methods. This semantic information is used to create conjectures to determine the commutativity properties of these methods. These conjectures are analyzed with the help of an automated theorem prover.

This paper describes the use of object semantics to determine the commutativity of groups of object method invocations. This information can then be used for maintaining the consistency of distributed object caches as well as for distributed concurrency control. Specifically, the main contributions of our work are the following.

- We introduce *method group* commutativity and describe how it may be used to support distributed object caching and concurrency control
- We present a formal methodology to extract commutativity from a semantic specification of an object. This methodology creates a *method commutativity specification* (MCS) that describes the conditions when several method invocations will commute. Our current framework uses the PVS theorem prover[10] to analyze and validate an object's commutativity properties.
- We extend this methodology by adding weaker consistency requirements to an object's semantic specification which permits the creation of an MCS with greater commutativity.

- We show how the use of this formal methodology can more precisely expose the commutativity properties than previous approaches. Our approach also unifies the commutativity represented by Weihl's *forward* and *backward* commutativity properties[16] in a single specification.

This paper presents these contributions in the following manner. Section 2 reviews previous approaches using object semantics and commutativity for concurrency control. Section 3 presents the motivation for the use of group commutativity. Section 4 describes a formal model for commutativity specification, analysis, and usage. We then illustrate, in Section 5, the results of our analysis by comparing them with results in the literature. Section 6 presents conclusions and future directions.

II. Related Work

A wealth of research about operation commutativity exists in the literature. In this section, we review prior research and show how that work has motivated and directed our work.

Some of the earliest work in operation commutativity was done by Schwarz[13] and Weihl[15] in their PhD theses. Describing his work, Schwarz, along with Spector[14], used the properties of several abstract data types, such as directory and queue, to determine lock compatibility tables between pairs of operations. This analysis was based on the determination of dependencies between types of operations, including the consideration of operation parameters. The generation of dependencies was generally based on read/write conflicts between components of the object. We will use their directory example to show how our approach results in a commutativity relationship with greater commutativity.

In his work, Weihl [16], [17], [18] identified commutativity relationships between pairs of operations. An operation was defined to include the parameters passed to the operation as well as the output of the operation. Weihl classified commutativity as two forms: *forward* and *right backward* commutativity. Informally, if two operations *forward* commute and are defined on a given state, then the effect of executing the operations in either order is the same. If operation β *right backward* commutes with operation α and it is possible to execute the operations in the order $\alpha\beta$, then executing the operations in the reverse order would have had the same effect. We will use Weihl's bank account example[17] to illustrate how our notion of commutative reflects, in a unified manner, both forward and backward commutativity.

Roesler and Burkhard [12] also exploit the commutativity of operations to create lock compatibility tables. Like our approach, they create a table for an object from its specification. Our approach differs in the manner in which the behavior of the object is specified and the manner in which commutativity is derived from that specification. In this paper, we use their queue example to illustrate cases where our approach using formal analysis reveals greater commutativity than their approach.

Other researchers have utilized object structure to determine commutativity and/or conflicts. Badrinath and Ramamritham [1] used the structure of objects as represented in granularity graphs to determine commutativity. Chrysanthis, Raghuram, and Ramamritham[3] used the classification of an operation as an modifier and/or observer to determine object lock compatibility in a transactional environment. Malta and Martinez[8], constructed commutativity matrices based on access modes of an attribute. While these approaches may capture some of the semantics of an object, they do not fully consider the formal semantic specification of the object in order to determine commutativity.

Muth, et. al, [9] analyzed the structure of objects in order to create compatibility matrices based on commutativity. This work was extended by Resende, Agrawal, and El Abbadi [11] to deal with shared object references. For our work, the complete semantic information about an object is reflected its top level semantic specification.

It may be possible, in some applications, to increase concurrency by weakening object semantics to realize greater commutativity. Schwarz and Spector[14] illustrated the benefits of weakening object semantics using a queue example. Wong and Agrawal [19] describe a technique that permits some bounded inconsistency to be viewed by an application. In this paper, we show how a similar effect can be achieved for method group commutativity. Also in the context of real-time applications, researchers have introduced schemes to increase concurrency by tolerating non-serializable executions if the potential errors are within certain bounds[5].

Showing that commutativity is still relevant, Wu and Fekete [20] illustrate how the use of commutativity, derived from either the structural read/write or semantic properties of the object, improved the performance of a telecommunications application. In their experience, semantics-based analysis of commutativity provides greater concurrency. Dennis, et. al, [4] illustrate how commutativity could be used during the requirements analysis of a system for radiation therapy.

III. Method Group Commutativity

Unlike traditional commutativity approaches which use pairwise commutativity relationships, our work is driven by a broader definition of commutativity, the *commutative method group*. This section introduces method group commutativity and outlines how method group commutativity may be used by a distributed application.

A. *Method Group Commutativity*

In our approach, we use method invocations on a object as the basis for our analysis. Rather than considering pairwise combinations of method invocations, we consider sets of method invocations for our analysis. We describe sets of method invocations having similar properties as a *method group*. Informally, a method group describes sets of method invocations that may be invoked on an object. To describe these sets, a method group has an associated group predicate, P_G, which is satisfied by the invocations in each set. For example, consider a *BankAccount* object which contains the *balance* of the account and three methods. The method *getBalance()* retrieves the *balance*. The *withdraw(p)* method returns OK and subtracts *p* from *balance*, when *p* is less than or equal to *balance*. The *deposit(p)* method adds *p* to *balance*. A simple method group could be defined as sets of method invocations, where each set consists of invocations of the *getBalance()* method. A more complex example is the set of all invocations of the *withdraw(p)* method, where the sum of all the parameters for the *withdraw(p)* methods is less than the current balance of the *BankAccount*. A formal definition of method groups is presented later in this paper.

We are interested in method groups where, for each possible set of method invocations, the method invocations in each set will commute. Informally, a method group is a *commutative method group* if, for each set of method invocations in the method group, executing these methods in the set, in any order, produces the same effect. The examples presented above are commutative method groups. The method group consisting of a *BankAccount*'s *getBalance()* methods is commutative because each method invocation will return the current balance of the object and will not change the current balance of the object. The previously defined method group consisting of the *BankAccount*'s *withdraw(p)* methods is also commutative because each method invocation will return OK and the resultant balance after the execution of the methods will be the same, regardless of the order in which those methods were executed.

B. *Uses of Method Group Commutativity*

Once commutative methods groups are defined, this information can be used to assure the consistency of method invocations in a distributed object caching environment. We briefly outline how this may be accomplished.

Consider an environment using the client/server model. The master copy of the object resides at the server, but the object is concurrently cached by remote clients. The server provides each client a semantic lock which indicates the methods which that client may execute on the object. This semantic lock assures that the object is accessed by the clients in a consistent manner. This is possible if the semantic locks only allow the clients to execute methods that commute with each other. A natural way to assigned these semantic locks is to permit clients to execute methods described by a commutative method group. Because of commutative properties, the computational effect will be the same as if the method was executed in any order on the master copy at the server.

In this environment, a middleware layer tracks the execution of the object methods and sends them to the server for execution on the master copy of the object. Again the order in which the updates are sent by the clients to the server does not matter since the methods commute. In such a setting, the middleware layer would determine which method group should be used to create the semantic lock. It then logically partitions the group predicate, P_G, into semantic locks that are given to the clients. The larger the number of possible method invocations described by the method group, the larger the amount of concurrent operations that may take place in the system.

IV. A Formal Model of Commutativity Specification, Analysis, and Usage

To exploit method group commutativity, we must be able to determine the commutative method groups. In this section, we introduce the Method Specification Table (MST), which describes the semantics of an object's methods. Next, we use the MST to create expressions which represent sequential execution of methods. We then provide a formal definition of commutativity in terms of the MST. This definition is used to provide a framework for commutativity analysis. Using this framework, a Method Commutativity Specification (MCS) is created. We have developed procedures using the PVS theorem prover that assist in commutativity analysis. Recognizing that weakening of consistency requirements may provide greater concurrency, an MCS may be modified to provide weaker consistency.

A. Method Specification Table

Our approach begins with a Method Specification Table (MST), which is a semantic specification of an object's methods. An MST consists of rows which describe the behavior of a method under specific preconditions. The columns of each row indicate the precondition (P), method name (M), state postcondition (S), and output postcondition (R). The precondition (P) is a boolean predicate of the form $f(T, p_1, ..., p_n)$, where T represents the state of the object and $p_1, ..., p_n$ represent the parameters of the method. One postcondition, the *state postcondition* (S), describes the effect of the method upon the state of the object. It has the form $T' = g(T, p_1, ..., p_n)$, meaning the new state, T', of the object is determined by the function g which uses the current state of the object and the parameters of the method. If object is not changed, then the expression is simply $T' = T$, or the identify function. The other postcondition, the *output postcondition* (R), describes the value returned by the method. This predicate has the form $V_i = h(T, p_1, ..., p_n)$, where V_i is the output value and the subscript i identifies the executed method. Because a method may have different behavior under different conditions, a method may be described by more than one row.

Rows of the MST may be viewed as a restricted form of Hoare Logic[6] expressions. A Hoare Logic expression is a triple, $P\,\{Q\}\,Z$, which describes the behavior of a program *Q* in terms of a precondition *P* and a postcondition *Z*. A row i of the MST, with columns P_i, M_i, S_i, R_i, may be considered a Hoare Logic expression where P_i is the Hoare Logic precondition, M_i is *Q* , and *Z* is the combination of S_i and R_i. For the remainder of the paper, we use the expression, $P_i\,\{M_i\}\,S_i \wedge R_i$, to represent a row of the MST.

An example of a MST for a *bank account* object is shown in Figure 1. The bank account has four methods whose behavior is described by five rows of the MST. In each row, the state of the object is represented by an integer, *bal*.

	P	M	S	R
1	TRUE	balance()	bal'= bal	$V_1 = bal$
2	x > bal	withdraw(x)	bal'= bal	$V_2 = FAIL$
3	y≤bal ∧ y > 0	withdraw(y)	bal' = bal - y	$V_3 = OK$
4	z > 0	deposit(z)	bal' = bal + z	$V_4 = OK$
5	TRUE	withdrawAll()	bal' = 0	$V_5 = bal$

Fig. 1. MST for Bank Account

Figure 2 shows the MST example for a "directory" or "hashtable." In this case, the behavior is defined in terms of logical *iskey()*, *put()*, *get()*, and *remove()* operations. The *iskey()* operation returns true if the key is present in the hashtable. We defined these logical functions in PVS using PVS's representation of a function. Figure 3 shows

#	P	M	S	R
1	a ≠NULL	put(w,a)	T'= put(T, w, a)	V_1=true
2	iskey(T, x)	get(x)	T'=T	V_2= get(T, x)
3	NOT iskey(T, y)	get(y)	T'=T	V_3=null
4	TRUE	delete(z)	T'= re-move(T,z)	V_4=true

Fig. 2. MST for Hashtable

the MST example for a FIFO queue. In this case, the behavior is defined in terms of logical *empty()*, *pushQentry()*, *popQentry()*, *getHead()*, and *printq()* operations. These queue operations, along with their names, were based on a PVS theorem describing queues that was found on the web[1].

[1] http://www.wisdom.weizmann.ac.il/~amir/Course02a/ queue.pvs

#	P	M	S	R
1	TRUE	enq(p1)	T' = push-Qentry (T,p1)	V_1 = true
2	NOT empty(T)	deq()	T'=pop-Qentry (T)	V_2 = get-Head(T)
3	empty(T)	deq()	T' = T	V_3 = FAILED
4	TRUE	printq()	T' = T	V_4 = printq(T)

Fig. 3. MST for Queue

B. Sequential Composition of Methods

The rows of the MST can be used to create expressions describing the behavior of the sequential execution of two methods. Consider the two Hoare Logic expressions representing rows i and j of an MST.

$$P_i\{M_i\}S_i \wedge R_i \quad and \quad P_j\{M_j\}S_j \wedge R_j$$

We now wish to create an expression that describes the behavior of executing the method sequence: M_i followed by M_j. To do this, we must determine the precondition that must be true before the method sequence. We also must determine the final state of the object and the values returned by each method.

First, we consider the precondition. Obviously, the precondition P_i must be true, since M_i is executed first. We then need to determine a precondition, which we identify as P_{ij}, that must be true before M_i executes such that P_j is true after M_i executes. Recall that P_j has the form $f_j(T, ...)$, where T is the state of the object before the execution of M_j. Since M_j is executed after M_i, the state T used by M_j, is the state produced by M_i, described by S_i. Because S_i is of the form $T' = g_i(T, ...)$, then the right side of the S_i expression can be substituted as the current state for P_j. Consequently, we derive that P_{ij} is $f_j(g_i(T, ...), ...)$. For the method sequence to be executed, both P_i and P_{ij} must be true.

Next, we must determine the final state of the object. Let the final state be presented as S_{ij} , meaning that state of the object after the execution of methods M_i and M_j. Similar to the evaluation of P_i, the state is derived by substituting the right side of the S_i expression into the state used by the S_j expression. Consequently, the final state of the object, S_{ij} is $T'' = g_j(g_i(T, ...), ...)$.

Finally, we must determine the values returned by each method. In this case, R_i, like P_i, remains the same. The value returned by M_j when it executes after M_i is R_{ij}. Like previous evaluations, this return value is derived using substitution to determine that R_{ij} is $V_j = h_j(g_i(T, ...), ...)$.

In summary, the following expression represents the precondition and postcondition of executing the two methods as a sequence, M_i, M_j, where P_{ij}, S_{ij}, and R_{ij} are derived as explained above.

$$P_i \wedge P_{ij}\{M_i;\ M_j\}S_{ij} \wedge R_i \wedge R_{ij}$$

Examining this expression, we can note the following. First, for the method sequence to be possible, the new precondition must be true for some states of the object and some parameters of the methods. If $P_i \wedge P_{ij}$ is never true, then it is not possible for the two methods (as described by the rows of the MST) to be executed in sequence. Second, in the trivial case where S_i does not change the state of the object, then $P_{ij} = P_j$, $S_{ij} = S_j$, and $R_{ij} = R_j$.

C. Commutativity Definition and Analysis

Once the method semantics are identified, we wish to identify the conditions under which groups of methods commute with each other. To do this, we first must have a definition of commutativity. While previous definitions currently exist [2][7][16], we present a definition based on the MST. Using this definition, we illustrate how commutativity relationships can be determined.

1) Commutativity Definition: For our commutativity definition, rather than considering pairs of methods that commute, commutativity is defined for a set of method invocations such that all method invocations in the set commute. Formalizing our previous definition, a *method group* identifies sets of method invocations from rows of the MST that satisfy a group precondition, P_G. This group precondition usually defines a relationship between the parameters of the methods in the method group and the state of the object. For example, consider the method group consisting of methods from row 3 of Figure 1, that satisfy the precondition $\sum i \leq 100$. Some possible sets of methods in this method group are the following: $\{withdraw(100)\}$, $\{withdraw(50), withdraw(50)\}$, and $\{withdraw(1), withdraw(1), withdraw(3)\}$.

We now wish to determine if a method group describes sets where, for every set in the method group, each set's method invocations commute with each other. If they do, then the method group is a *commutative method group*.

Consider a method group with a group precondition, P_G, which defines sets of methods, where each set, M, consists of methods $m_1, m_2, ..., m_n$. This method group is a *commutative method group* if for every state of the object and every set M permitted by P_G, and for all possible orderings of $m_1, m_2, ..., m_n$, the following conditions are true.:

- CC1 (Commutativity Condition 1) :
  The precondition for each method in set M is always true before the method's execution, regardless of the order in which it is executed.
- CC2 (Commutativity Condition 2):
  The final state of the object is the same for all possible orderings of $m_1, m_2, ..., m_n$.
- CC3 (Commutativity Condition 3):
  Each method's return value remains the same, regardless of the order in which it is executed.

2) Using Hoare Logic to Analyze Commutativity: Given these three commutativity conditions, we can use the Hoare Logic representation of rows of the MST to determine whether those rows commute. We first consider the sequential execution of two methods. After determining the pair-wise commutativity of methods, the analysis is extended to larger groups of methods.

For two methods M_i and M_j, the preconditions and postconditions for the two possible method sequences are the following:

$$P_i \wedge P_{ij}\{M_i;\ M_j\}S_{ij} \wedge R_i \wedge R_{ij} \qquad P_j \wedge P_{ji}\{M_j;\ M_i\}S_{ji} \wedge R_j \wedge R_{ji}$$

The two methods commute if they satisfy the three commutativity conditions. First, because of the definition of P_{ij} and P_{ji}, CC1 will alway be satisfied when $P_i \wedge P_{ij} \wedge P_j \wedge P_{ji}$ is true. Let P_R represent this required precondition ($P_i \wedge P_{ij} \wedge P_j \wedge P_{ji}$). When choosing a group precondition, P_G, it must be chosen such that $P_G \supset P_R$. In many cases P_G can be chosen to be P_R.

Second, to satisfy CC2, the final state of the object for the two sequences must be the same when the precondition, P_G, is true. If S_{ij} is of the form, $T' = g_j(g_i(T, ...), ...)$, and S_{ji} is of the form, $T' = g_i(g_j(T, ...)$
$, ...)$, then the states are the same when $g_j(g_i(T, ...), ...) = g_i(g_j(T, ...), ...)$. This is logically equivalent to stating that $S_{ij} \wedge S_{ji}$ is true.

Third, to satisfy CC3, the return values must be equal. If R_i is of the form, $V_i = h_i(T, ...)$, and R_{ji} is of the form, $V_i = h_i(g_j(T, ...))$, then the return values are equal when $h_i(T, ...) = h_i(g_j(T, ...))$. In other terms, $R_i \wedge R_{ji}$ is true when the precondition is true. Similarly, $R_j \wedge R_{ij}$ must also be true.

These three conditions can be combined into logical conjectures which indicate if the methods will or will not commute given the precondition. These conjectures are analyzed for all states of the object and for all parameters of the methods. These conjectures are the following.

Conjecture 1: $P_R \supset NOT\ (S_{ij} \wedge S_{ji} \wedge (R_i \wedge R_{ji}) \wedge (R_j \wedge R_{ij}))$
Conjecture 1 is used to determine if the two methods do not commute. Two methods do not commute if the precondition is always false. Otherwise, if the precondition can be true, then the methods do not commute since one of the postconditions is false.

Conjecture 2: $P_R \supset (S_{ij} \wedge S_{ji} \wedge (R_i \wedge R_{ji}) \wedge (R_j \wedge R_{ij}))$
If it can be shown that the combined precondition P_R is not always false, then Conjecture 2 can be used to determine if the methods commute. This conjecture requires that when the precondition is true, then the postconditions are true.

It may be possible that neither of these conjectures can be proved. For example, there may be some cases where CC2 and CC3 may be satisfied for some cases where the precondition is true. If this is the case, it may be necessary to add an additional precondition, P_s, to assure that $R_i \wedge R_{ji}$ and $R_j \wedge R_{ij}$ and $S_{ij} \wedge S_{ji}$ are always true when the precondition is true. In this case, the group precondition P_G is be chosen to be $P_s \wedge P_i \wedge P_{ij} \wedge P_j \wedge P_{ji}$. We can then use the following conjectures to determine if the rows will commute given a group precondition, P_G.

Conjecture 3: $P_G \supset (S_{ij} \wedge S_{ji} \wedge (R_i \wedge R_{ji}) \wedge (R_j \wedge R_{ij}))$

First, Conjecture 3 is used to show that CC2 and CC3 are satisfied when the group condition is true.

Conjecture 4: $P_G \supset (P_i \wedge P_{ij} \wedge P_j \wedge P_{ji})$

Second, Conjecture 4 is used to verify that the group condition is stricter than the precondition required by CC1.

Conjecture 5: $P_R \wedge (\neg P_G) \supset \neg(S_{ij} \wedge S_{ji} \wedge (R_i \wedge R_{ji}) \wedge (R_j \wedge R_{ij}))$

Given that we have a group precondition, P_G, which is stronger than the calculated precondition, Conjecture 5 can be used to verify that the group precondition is *complete*. A group precondition is complete if it is the weakest group condition that will permit commutativity. If there exists a case where P_R is true but P_G is false and the methods still commute, then it will not be possible to prove this conjecture. As with other conjectures, the processing of attempting the proof may reveal a clue as to what condition may need to be added to P_G in order to assure completeness.

The creation of these conjectures is easily extended to the consideration of larger groups of method invocations. For example, a group of three methods would result in six possible orders, instead of two. This results in larger expressions that are used for the preconditions and postconditions. While these conditions are cumbersome to manually create, their creation is easily automated.

D. Method Commutativity Specification

Unlike previous approaches which create a matrix to specify which pairs of methods commute, we create a method commutativity specification (MCS) which identifies commutative method groups. Each row of the MCS identifies a commutativity method group. Each row identifies the rows of the MST which the methods come from, the methods that commute, and the condition under which the methods commute. In the MCS, we use the following notation to indicate the grouping of a number of method invocations of methods m_i and m_j: $\{m_i, m_j\}$*. Figure 4 shows a portion of the MCS for a bank account. Examining row 1, we see that this method group was created from row 1 of the MST. It consists of the balance() method and any number of its invocations will always commute. Row 7 shows the methods from rows 1 and 4 of the MST. These methods, *balance()* and *withdraw(x)*, have a group condition of FALSE. A FALSE condition indicates that the methods, as described by rows of the MST, do not commute. The method group with the most methods is shown in row 11. This method group was created from rows 2, 3, and 4 of the MST. The methods are *withdraw(x)* and *deposit(z)*. The condition when the methods commute depends upon the value of the withdraw parameter, *x*. If *x* is greater than *bal*, then it must be greater than *bal* plus the sum of all the deposit parameters, *z*. Otherwise, if *x* is less than *bal*, then the sum of all *x* parameters (less than *bal)* must be less than or equal to *bal*. The methodology for creating the MCS is described below.

The MCS is built incrementally. First, method groups consisting of method invocations from one row of the MST are considered. For the bank account object, these are shown in rows 1 through 4 in Figure 4. Then method groups consisting of methods from two rows of the MST are considered. The process is continued until all possible groupings have been considered.

MCS Row	From MST Rows	Methods	Group Conditions
1	1	{balance()}*	TRUE
2	2	{withdraw(x)}*	$\forall x$ x > bal
3	3	{withdraw(y)}*	$\forall y \sum y$ < bal
4	4	{deposit(z)}*	TRUE
5	1,2	{balance(), withdraw(x)}	$\forall x$ x > bal
6	1,3	{ balance(), withdraw(y)}*	FALSE
7	1,4	{balance(), deposit(z)}	FALSE
8	2,3	{withdraw(x)}	$\forall x\, where\, x > bal, true,\ otherwise \sum x < bal$
9	2,4	{withdraw(x), deposit(z)}*	$\forall x,\ x > bal + \sum z$
10	3,4	{ withdraw(y) , deposit(z)}*	$\sum y\ \leq$bal
11	2,3,4	{withdraw(x), deposit(z)}*	$\forall x\, where\, x > bal + \sum z,\ true,\ otherwise \sum x < bal$

Fig. 4. MCS (Method Commutativity Specification) for Bank Account

If n is the number of rows in the MST, then at most $2^n - 1$ combinations of methods must be considered to generate a complete MCS. However, if it is determined that a group of methods does not commute, other method groups containing that method group do not need to be considered. For example, the method group consisting of rows 1,2,3 of the MST of the bank account object is not considered because rows 1 and 3 of the MST do not commute, as shown by row 6 of Figure 4.

When generating each row of the MCS, the following steps are taken.

1) Determine if Conjecture 1, which implies *non commutativity*, is satisfied. For example, rows 1 and 3 in Figure 1 do not commute because $R_{13} \wedge R_1$ is always false.
2) Otherwise, determine if Conjecture 2, which implies commutativity, is satisfied. For example, rows 1 and 2 in Figure 1 commute.
3) Otherwise, choose a group precondition that is a stricter than the precondition. Typically, the analysis of Conjecture 2, using a theorem prover, generally provides hints as to how the precondition should be modified. Then determine if Conjecture 3 and Conjecture 4 are true. If so, the methods commute under precondition P_G. If they do not commute, then this step is repeated until a valid group precondition is determined. For example, the entry in the MCS corresponding to the analysis of the withdraw method is shown in row 3 of Figure 4.
4) We can also use Conjecture 5 to determine if the chosen P_G is complete. Choosing a P_G that is complete will verify that no further refinement of P_G is necessary because there does not exist any P_X where $P_G \supset P_X \supset P_R$, and the methods commute when P_X is true.

E. Tool Support

To assist in the commutativity analysis, we have created a procedure to iteratively build an MCS from an MST. The procedure is driven by a human analyst, assisted by appropriate tools. We developed a tool to create appropriate conjectures that are proved with the aid of the PVS theorem prover. The purpose of this tool is to assist in the analysis of commutativity and in the validation of the MCS.

The procedure to build the MCS is shown in Figure 5. On the first pass, the tool, the AnalyzeMST, reads the MST and creates an empty MCS. AnalyzeMST uses this information to generate PVS conjectures about the commutativity of the methods. The PVS theorem prover is then used to prove if the commutativity conjectures are correct. If the conjectures can be proved, then AnalyzeMST updates the MCS with the appropriate information. If a conjecture cannot be proved, then the output of the PVS analysis is saved. This output can be examined by the analyst to determine the changes to be made to the MCS. The analyst then updates the MCS and repeats the process until the analysis is complete.

The first time AnalyzeMST is used, an empty MCS is created. For each combination of rows, an empty MCS indicates that the commutativity is unknown. When encountering a combination of rows where the commutativity is unknown, AnalyzeMST first attempts to prove that the rows do not commute using Conjecture 1. If this can be proved, the MCS is updated with that information. If it cannot be proved, then AnalyzeMST attempts to prove that the methods commute, using Conjecture 2. If the conjecture can be proved, then AnalyzeMST will then attempt to prove that the combination precondition, P_R, is TRUE. The purpose of this step is to show that the precondition is always true. If the precondition is always true, then the MCS is updated to indicate that the methods always commute. If the precondition is not always true, then the resulting PVS output will contain a simplification of the precondition. The analyst then uses this information to update the MCS with the appropriate precondition for the row.

If Conjecture 2 is not true, meaning the rows do not commute for P_R, the resultant PVS output may contain clues as to which conditions may need to be added to the MCS. Using this information, the analyst updates the MCS with appropriate precondition, P_G, for the row.

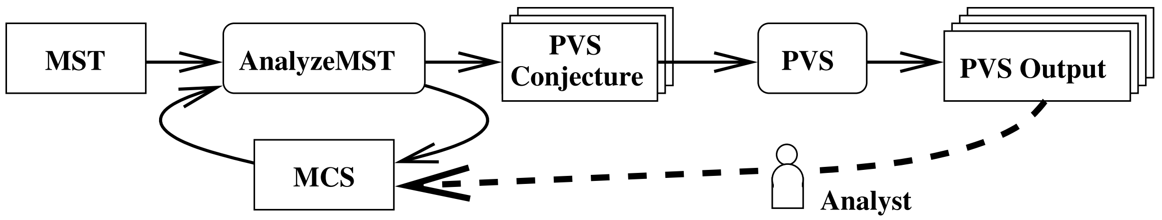

Fig. 5. Analysis Process

Row	From Rows	Methods	Weakening Condition	Conditions
6	1,3	{ balance(), withdraw(y)}*	return(1, 100)	$\forall y \sum y < 100$
7	1,4	{balance(),deposit(z)}*	return(1, 100)	$\forall z \sum z < 100$
12	1,3,4	{ balance(), withdraw(y), deposit(z)}*	return(1, 100)	$\forall y, z \sum y + \sum z < 100$

Fig. 6. Weakened MCS for Bank Account

On subsequent uses of AnalyzeMST, each combination of rows will contain an indication that the rows do not commute or that the rows commute under a certain precondition, P_G. Note that P_G may be the TRUE condition, meaning the rows always commute. When P_G is specified, AnalyzeMST attempts to prove that the rows commute using Conjecture 3. If this conjecture cannot be proved, then the appropriate row of the MCS is updated with an indication that the commutativity is unknown. At this point, the analyst has entered incorrect information for the MCS and the MCS must be updated with a different condition. Otherwise, if Conjecture 3 was proved, then AnalyzeMST attempts to prove Conjecture 4, which states the P_G must be stronger than P_R. If this cannot be proved, then the precondition, P_G, must be reworked. Finally, AnalyzeMST then attempts to prove Conjecture 5 to verify that P_G is complete. If this cannot be proved, the proof analysis may contain a clue as to which condition may need to be added to P_G. As mentioned early, if it is possible to prove Conjecture 5, then we have proved that we have identified all the conditions for that row which permit commutativity.

The procedure is repeated until it can be shown the that conditions specified in the MCS are correct and complete.

F. Weakening of Consistency

As mentioned earlier, several researchers noted that weakening the semantics of operations can lead to greater commutativity. In our framework, we can weaken the consistency requirements by changing the Commutativity Conditions discussed in Section IV-C.1. In this subsection we consider how CC3 (Commutativity Condition 3) may be weakened.

Recall that CC3 requires that each method's return value remains the same, regardless of the order in which the method is executed. We can weaken CC3 by permitting the return value to differ by some degree of imprecision. Instead of requiring the return values of a method for different sequences to be equivalent, we can require that they satisfy a certain function. For example, suppose that our imprecision function is *weakReturn*, then we can rewrite Conjecture 3 as the following Conjecture .

Conjecture 6: $P_G \supset (S_{ij} \wedge S_{ji} \wedge$
$weakReturn(R_i, R_{ji}) \wedge weakReturn(R_j, R_{ij}))$
We have adopted the MCS and the analysis tools to use a weakening condition which specifies the inconsistency that is allowed for the return value. For example, we applied weakening to the bank account example such that any value returned by the *balance* method will be within 100 of the value that would have been returned for a different method invocation sequence. To accomplish this, we added a column to the MCS to indicate a weakening condition to be used during the analysis. For our analysis, the condition, *return(1,100),* represents that the imprecision of the value returned by the method in MST row 1 is less than 100. When encountering a weakening condition, the analysis tool will use Conjecture 6 instead of Conjecture 3. The analysis procedure is then used as described above. Applying this analysis, we obtained new rows in the MCS for the Bank Account example as shown in Figure 6. Because of the permitted imprecision, Row 6 shows that the *balance()* and *withdraw()* methods will commute as long as the total amount withdrawn is less than 100. Similarly, Row 7 shows that the *balance()* and *deposit()* methods will commute as long as the total amount deposited is less that 100. Row 12 shows that all three methods will commute as long as the sum of the parameters to *withdraw()* and *deposit()* is less than 100.

In this MCS, we note that the weakened condition lead to conditions that constrained which *withdraw()* and *deposit()* methods were permitted to execute. The constraints on these methods assure that the consistency conditions would not be violated.

V. Analysis Results

In this section, we compare the results of analyzing object methods using our methodology to previous results in the literature. We consider the following objects: directory or hashtable, bank account, and queue.

A. *Directory / Hashtable*

The first example we consider is the directory from Schwarz and Spector[14]. A directory represents a mapping from a key to a value. A $Modify(\sigma)$ operation is used to associate a value with a key and a $Lookup(\sigma)$ operation is used to retrieve the value associated with a key. The lock compatibility table from their paper is shown in Figure 7. In this table, a "No" entry indicates that the operations conflict. Note that a $Modfy(\sigma)$ operation conflicts with a $Modify(\sigma)$ operation and a $Lookup(\sigma)$ operation where the key parameters, σ, are the same.

	Lock Held	Lock Held
Lock requested	$Modify(\sigma)$	$Lookup(\sigma)$
$Modify(\sigma)$	**No**	**No**
$Modify(\sigma')$	OK	OK
$Lookup(\sigma)$	**No**	OK
$Lookup(\sigma')$	OK	OK

Fig. 7. Directory Lock Compatibility Table

We evaluated a hashtable, which is functionally equivalent to the directory. The MST for the hashtable was previously shown in Figure 2. In our analysis, the $put()$ method corresponds to the $Modify()$ operation and the $get()$ method corresponds to the $Lookup()$ operation. After applying our methodology, we obtained the MCS shown in Figure 8.

MCS Row	From MST Rows	Methods	Conditions
1	1,1	{put(w,a)}*	$\forall w_1, ..., w_n\, if\, (i \neq j \wedge w_i = w_j)\, then\, a_i = a_j$
2	1,2	{put(w,a),get(x)}*	$\forall w_1, ..., w_n\, if\, (i \neq j \wedge w_i = w_j)\, then\, a_i = a_j \wedge \forall x,\, iskey(x) \wedge \forall w, x,\, if\, x = w\, then\, a = get(x)$
3	1,3	{put(w,a),get(y)}*	$\forall w_1, ..., w_n\, if\, (i \neq j \wedge w_i = w_j)\, then\, a_i = a_j \wedge \forall y,\, \neg iskey(y) \wedge \forall w, x\, w \neq x$
4	2,3	{get(x)}*	TRUE

Fig. 8. Partial MCS (Method Commutativity Specification) for Hashtable

If we express the MCS as a table with two parameters with each entry representing the conditions where the operations commute, we obtain Figure 9. Comparing Figure 9 to Figure 7, we see that our method has detected the same OK cases as the previous results. However, because of our completeness test, we have also determined that a *put(w,a)* method commutes another *put(w,b)* method if the value parameters, a and b, are identical. Furthermore, we determined that a *put(w,b)* method commutes with a *get(w)* method if the new value is identical to the old value. While this may be intuitively obvious, these results were obtained by refining the MST until the completeness test for Conjecture 5 was passed.

	$put(w, a)$	$get(w)$
$put(w, b)$	**a=b**	**a=get(w)**
$put(w', b)$	OK	OK
$get(w)$	**a=get(w)**	OK
$get(w')$	OK	OK

Fig. 9. Hashtable Pairwise Commutativity

B. *Bank Account*

The next example that we consider is the bank account. In Section 2, we reviewed Weihl's definitions of forward and backward commutativity. Figures 10 and 11 show the commutativity tables, from [17], for a bank account for forward and right backward commutativity. In these tables, D,W,B represent deposit, withdraw, and balance respectively. In these tables, we have labeled the entries we wish to discuss as (1),(2),(3),(4), and (5).

	[D(j),ok]	[W(j),OK]	[W(j),NO]	[B,j]
[D(i),ok]	OK	OK	**NO (1)**	NO
[W(i),OK]	**OK (2)**	**NO (3)**	OK	NO
[W(i),NO]	**NO (4)**	**OK (5)**	OK	OK
[B,i]	NO	NO	OK	OK

Fig. 10. Forward Commutativity for Bank Account

	[D(j),ok]	[W(j),OK]	[W(j),NO]	[B,j]
[D(i),ok]	OK	OK	**NO (1)**	NO
[W(i),OK]	**NO (2)**	**OK (3)**	OK	NO
[W(i),NO]	**OK (4)**	**NO (5)**	OK	OK
[B,i]	NO	NO	OK	OK

Fig. 11. Right Backward Commutativity for Bank Account

As shown by entry (1) in these tables, a *withdraw* operation does not forward or right backward commute with a *deposit* operation. They do not commute because a *deposit* operation could cause a subsequent *withdraw* operation to return *ok* instead of *NO*. As shown by entries 2 and 5, there are some cases where operations forward commute, but they do not right backward commute. Conversely, entries 3 and 4 show cases where the operations right backward commute, but they do not forward commute.

Shown earlier in Figure 4 is the MCS for the bank account object. For comparison purposes, in Figure 12 we represent the MCS in the same form as Weihl's table. If the methods do not always commute, an entry in this table contains a condition, derived from the MCS, that indicates when the methods commute.

	[D(j),ok]	[W(j),OK]	[W(j),NO]	[B,j]
[D(i),ok]	T	$j \leq bal$	$j >$ $bal + i$	**NO**
[W(i),OK]	$i \leq bal$	$i + j$ $\leq bal$	$j > bal \wedge$ $i \leq bal$	**NO**
[W(i),NO	$i > bal$ $+j$	$i > bal \wedge$ $j \leq bal$	$i > bal \wedge$ $j > bal$	T
[B,i]	**NO**	**NO**	T	T

Fig. 12. Pairwise Results of Analysis

Comparing Figure 12 to Figures 10 and 11 we make the following observations. For entry 1, our analysis detected that [withdraw(j), NO] will commute with [deposit(i), ok] as long as the $j > bal + i$ condition holds. In this case our analysis has exposed commutativity not present in both forward and right back commutativity. For the other 4 cases, our analysis has detected commutativity that is present in either forward or backward commutativity, but not both.

This example shows that our approach encompasses both notions of forward and backward commutativity. It has detected commutativity not present in either form of commutativity as well as detecting commutativity present with either forward or backward commutativity. This demonstrates that our analysis unifies both these notions by specifically stating the conditions when the methods will commute.

C. Queue

The next example that we consider is the FIFO queue example from Roesler and Burkhard[12]. Figure 13 show their compatibility table. The queue is defined to have the standard FIFO queue operations such as enqueue(*enq*) and dequeue (*deq*). It also includes an output to display the contents of the queue (*printq*). In this table, note that their analysis shows that $enq()$ will not conflict another $enq()$ when they place the same value on the queue. Also note that they indicate that *deq()* will always conflict with another $deq()$.

	<enq(g), true	<deq, g>	<deq, no>	<printq, q>
<enq(g'),true>	$Y_{g=g'}$	Y	N	N
<deq, g'>	Y	**N**	x	N
<deq,no>	N	x	Y	Y
<printq,q>	N	N	Y	Y

Fig. 13. Queue Compatibility Table from Roesler and Burkhard

For our analysis, we use the MST that was presented earlier in Figure 3. Using our analysis, we generated an MCS. Portions of that MCS are shown in Figure 14. Row 1 of the MCS shows, like Figure 14, that *enq()* will commute with itself as long as the same element is enqueued. This is the same result as was found by Roesler and Burkhard. On the other hand, row 2 of the MCS shows that *n* invocations of *deq()* will commute as long as the top *n* elements of the queue are identical. The remaining rows of the MCS show results that are equivalent to previous results.

Row	From Rows	Methods	Conditions
1	1,1	{enq(p)}*	$\forall p_1, ..., p_n \, p_1 = p_x$
2	**2,2**	**{deq()}***	$count(deq) \leq size() \wedge$ $"top\, count(deq)\, elements\, of\, queue\, are\, equal"$
3	3,3	{deq()}*	$queue\, is\, empty$
4	4,4	{printq()}	TRUE
5	1,2	{enq(p), deq()}	$\forall p_1, ..., p_n \, p_1 = p_x count(deq) \leq size() \wedge$ $"top\, count(deq)\, elements\, of\, queue\, are\, equal"$
6	1,3	{enq(p), deq()}*	FALSE
7	1,4	{enq(p), printq()}*	FALSE
8	2,3	{deq()}*	FALSE
9	2,4	{deq(),printq()}*	FALSE
10	3,4	{deq(),printq()}*	$queue\, is\, empty$

Fig. 14. Queue MCS

During our analysis, we initially missed the case where the *deq()* method commutes with itself. Only by using the completeness condition, as expressed by Conjecture 5, were we able to determine that a condition was missing. Using the output of the PVS analysis as a clue, we were able to determine the additional condition that the methods will commute when the items they dequeue are identical. While this condition makes sense intuitively, we were able to find the condition using our formalized analysis.

VI. Conclusion

In this paper, we have introduced the *method commutative specification* (MCS) which defines when a group of method invocations commute. This new approach identifies sets of method invocations that commute, in contrast to existing approaches that only consider pairs of method invocations. We also presented a formal methodology, based on Hoare Logic, to create the MCS from the semantic specifications of an object. We have also illustrated how weakened consistency requirements may be added to these specifications, which then results in an MCS that permits greater commutativity. We have also shown how this approach can expose greater commutativity than current approaches in the literature. We have also shown how our analysis identifies, in a single representation, the commutativity present in both forward and backward commutativity.

This paper serves as the foundation for our ongoing work on distributed object management and caching. The weakening approach presented in this paper used a simple value based weakening of a method's return value. We are currently refining our analysis to permit the use of both version-based and time-based weakening. Another way to expose more commutativity is to weaken the other commutativity conditions, CC1 and CC2. We are currently investigating the weakening of these conditions.

We are also implementing a prototype that will use the semantic information specified in an MCS at runtime. This will be used to illustrate the usefulness of group commutativity concepts in managing cached and replicated copies of objects in distributed environments. We anticipate that group commutativity will be particularly useful when coupled with real-time and mobile applications.

References

[1] Badrinath, B. R., and Ramamritham, K. Synchronizing transactions on objects. *IEEE Transactions on Computers 37*, 5 (May 1998), 541–547.

[2] Beeri, C., Bernstein, P. A., Goodman, N., Lai, M., and Shasha, D. E. Concurrency control theory for nested transactions. In *Proc. of the Second ACM Symp. on Principles of Distributed Computing* (Montreal, Aug. 1983), pp. 45–62.

[3] Chrysanthis, P. K., Raghuram, S., and Ramamritham, K. Extracting concurrency from objects: A methodology. In *Proceedings of the 1991 ACM SIGMOD int'l conf. on Management of data* (Denver, Colorado, USA, May 1991), pp. 108–117.

[4] Dennis, G., Seater, R., Rayside, D., and Jackson, D. Automating commutativity analysis at the design level. In *Proceedings of ISSTA'04* (Boston, Massachusetts, USA, July 2004).

[5] DiPippo, L. B. C., and Wolfe, V. F. Object-based semantic real-time concurrency control. In *Proc. IEEE Real-time Systems Symposium* (Dec. 1993), pp. 87–96.

[6] Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM 12*, 10 (Dec. 1969), 576–580,583.

[7] Korth, H. F. Locking primitives in a database system. *Journal of the ACM 30*, 1 (Jan. 1983), 55–79.

[8] Malta, C., and Martinez, J. Automating fine concurrency control in object-oriented databases. In *Proceedings of the Ninth IEEE Conference on Data Engineering* (Vienna, Austria, Apr. 1993), pp. 253–260.

[9] Muth, P., Rakow, T. C., Weikum, G., Brössler, P., and Hasse, C. Semantic concurrency control in object-oriented database systems. In *Proc. of the 9th IEEE int. Conf. on Data Engineering* (Washington, D.C., Apr. 1993), pp. 233–242.

[10] Owre, S., Rushby, J., Shankar, N., and Stringer-Calvert, D. PVS: an experience report. In *Applied Formal Methods—FM-Trends 98* (Boppard, Germany, 1998), pp. 338–345.

[11] Resende, R. F., Agrawal, D., and Abbadi, A. E. Semantic locking in object-oriented database systems. In *OOPSLA'94* (Portland, OR, Oct. 1994), pp. 388–402.

[12] Roesler, M., and Burkhard, W. A. Concurrency control scheme for shared objects: A peephole approach based on semantics. In *Proceedings of the 7th International Conference on Distributed Computing Systems* (Washington, DC, USA, Sept. 1987), pp. 224–231.

[13] Schwarz, P. M. *Transactions on Typed Object*. PhD thesis, CMU, Dec. 1984.

[14] Schwarz, P. M., and Spector, A. Z. Synchronizing shared abstract data types. *ACM Trans. Computing Systems 2*, 3 (Aug. 1984), 223–250.

[15] Weihl, W. E. *Specification and Implementation of Atomic DataTypes*. PhD thesis, MIT, Mar. 1984.

[16] Weihl, W. E. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers 37*, 12 (Dec. 1988), 1488–1505.

[17] Weihl, W. E. The impact of recovery on concurrency control. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems* (Mar. 1989), pp. 259–269.

[18] Weihl, W. E. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions and Programming Languages and Systems 11*, 2 (Apr. 1989).

[19] Wong, M. H., and Agrawal, D. Tolerating bounded inconsistency for increasing concurrency in database systems. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of Database Systems* (San Diego, California, United States, 1992), pp. 236–245.

[20] Wu, P., and Fekete, A. An empirical study of commutativity in application code. In *Seventh International Database Engineering and Applications Symposium (IDEAS'03)* (Hong Kong, SAR, July 2003), p. 358.