Semantics-based Time Warp Protocols*

Hong Va Leong[†] Divyakant Agrawal[†]
Department of Computer Science
University of California
Santa Barbara, CA 93106

Abstract

The time warp mechanism is one of the useful paradigms to carry out a distributed discrete event simulation, where the synchronization of events can be determined a priori when the program is being executed. The forward computation performed optimistically has to be rolled back when an out-of-order synchronization message is received. This excessive amount of rollback seriously jeopardizes the performance of a time warp mechanism. Semantics of the synchronization messages being sent in a time warp execution can be exploited to reduce the amount of rollback. In particular, the commutativity and dependency properties of messages can be used to avoid certain types of rollback. For example, two commutative messages can be processed in any order and the resultant computation remains valid. In this paper, the semantics on the messages are defined and the protocols for handling such kinds of messages in the time warp mechanism and several optimizations are presented.

Keywords: message semantics, commutativity, distributed discrete event simulation, time warp, rollback

---

*An abridged version of this paper appears in the *Proceedings of the 7th International Workshop on Distributed Algorithms* as Using Message Semantics to Reduce Rollback in the Time Warp Mechanism.

[†]This research is supported by the National Science Foundation under grant number IRI-9117094.

1 Introduction

Synchronization of parallel programs in a message passing system is an important research topic that crosses several disciplines in computer science: distributed computing, computer architectures, and programming languages. Several synchronization mechanisms and models have been proposed to simplify the design of parallel or concurrent programs [CM88, BJKS88, LH89, BND89, ABHN91]. In general, the main source of complexity in synchronizing parallel programs is due to the non-deterministic nature of interactions among processes in the parallel programs. The non-determinism is inherent in such environments due to the asynchrony of message passing systems and due to the uncertainty of the relative speeds of process execution in parallel programs.

For a class of parallel programs corresponding to distributed discrete event simulation [Mis86, Fuj90], the synchronization complexity can be greatly reduced by eliminating non-determinism of execution order. In particular, since events in a distributed discrete event simulation are determined a priori, the ordering constraints of events can be enforced internally instead of relying on the real time execution of the parallel program to determine the ordering of events. Jefferson [Jef85] proposed the notion of *virtual time* for synchronizing processes in distributed discrete event simulation. Unlike real time, virtual time is not a monotonically increasing, single-valued function, but rather it allows different portions of a computation to advance and retract their own relative times, dynamically preserving the precedence constraints. The *Time Warp Mechanism* (TWM) [Jef85] is an implementation of virtual time in a message passing environment, in which the processes of a parallel program are synchronized using partial *rollback* instead of the conventional method of blocking [CM79].

The TWM synchronizes a set of processes corresponding to a parallel program or a distributed simulation over a message passing system. Each process has a logical clock that is used to maintain the virtual time at that process. The processes communicate with each other via messages that have two timestamps associated with them: *send virtual time* is the virtual time when the message was sent from its source and *receive virtual time* is the virtual time at which the message must be processed at its destination. Furthermore, each process maintains a queue of its internal state after processing each input message, a queue of input messages ordered by the receive virtual time, and a queue of copies of output messages ordered by the send virtual time. Processes execute optimistically by processing input messages and by advancing their local virtual time to the receive virtual time of the processed message without any regard to synchronization conflicts with other processes. A synchronization conflict arises in the TWM when an input message arrives at a process with a smaller receive virtual time than the local virtual time of that process. As a result, the process has to rollback to the virtual time smaller than the receive virtual time of the out-of-order message. This rollback is accomplished by restoring an older state of the process and by withdrawing (annihilating) all output messages that were sent after the rollbacked virtual time. The annihilation of output messages may result in the rollback of other processes. Jefferson [Jef85] has shown that despite such rollbacks, there exists a nondecreasing systemwide virtual time called *global virtual time*, such that no process will rollback below the current global virtual time.

One of the main drawbacks of the TWM is the phenomenon of cascading rollback, i.e., rollback at a process causing rollback at other processes and so on. Clearly, since the inherent synchronization technique in the TWM is based on rollback, there is no hope of completely eliminating the problem of cascading rollbacks. However, it is still advantageous to develop optimizations, that can be used to minimize rollback in the TWM, resulting in better performance. Rollback cost is not high in practical systems, but the cost of wasted forward computation due to rollback is extremely high [Fuj90]. It is therefore important to reduce rollback distance and limit the effect of a rollback. Several optimizations have been suggested to address the problem of rollback in the TWM such as limiting the forward progress of the computation as in the Moving Time Windows scheme [SSH89] or by limiting the propagation of messages [DR90, Ste91]. Another popular technique is called *lazy cancellation* of output messages [Gaf88]. In lazy cancellation, output messages are not annihilated as soon as a process rolls back. Instead, these messages are cancelled lazily (if necessary[1]) during the forward re-execution of the process. Furthermore, the technique of *lazy re-evaluation* has also been proposed [Wes88]. The hope is that at a certain point in the forward re-execution, the new process state will become identical to the prior (incorrect) state rendering annihilation of all subsequent output messages as well as the re-execution of previously processed input messages unnecessary. Sokol, Stucky and Hwang [SSH89] have proposed the Moving Time Window system which employs the notions of *non-side-effecting* and *side-effecting* messages (corresponding to the read and write operations). The arrivals of out-of-order non-side-effecting messages do not necessitate a rollback, since they do not change the object-state. Bagrodia and Liao [BL92] introduced "associative" messages that can be used with TWM. Their notion of association is commutative in nature and an associative sequence of messages guarantees the same final state and the same set of output messages whenever it is permuted, but detecting the sequence is extremely difficult, if not impossible. In particular, the authors reduce "artificial" rollbacks in a very common simulation object: the priority server. They distinguish between the receive time of a message (server request) and the earliest time the message can be served (when server may become idle). Rollback is avoided when the current time is smaller than the earliest possible service time, even though the receive time is smaller than the current time.

In this paper, we propose to implement (some of) the processes in the TWM as objects derived from abstract data type specifications, such as queues, sets and matrices. As a result, messages destined to these objects can be interpreted as operation invocations whereas messages sent out from these objects can be interpreted as operation responses. The type-specific properties, e.g., the *commutativity* relation [Kor83, SS84, Wei89] among operations, of the objects in the TWM can now be used to alleviate the rollback problem. Informally, we say that two operations *commute* if the resultant state is independent of the order of execution of the two operations. As an example, consider an object in a TWM based distributed simulation that is derived from a *set* abstract data type. Consider the case when the set object has executed an *insert* operation at time 15 and subsequently another insert operation with time 10 arrives at this object. In contrast to the

[1] The content of the old output message is checked against its new counterpart. If the two are identical, there is no need to annihilate the old output message and send the new one.

traditional TWM mechanism, we can now use the semantics of the insert operation to determine whether the insert at time 15 needs to be rolled back. It turns out that the result will be identical, even if the out-of-order insert is executed after the insert operation at time 15. Reducing the rollback distance cuts the rollback cost, and more importantly, reduces the forward recomputation and the resulting cascading rollback brought about by unnecessary cancellation of the "incorrect" forward computation.

The paper is organized as follows. We first review several type-specific properties and motivate their use with the help of an example showing how these properties can be used to minimize rollback in the TWM. Next, we develop the formal model of the system and state the rules for employing type-specific properties for minimizing rollback in the system. In Section 4, we present a simple protocol to implement a semantics-based time warp mechanism. New type-specific properties are defined and integrated into existing properties in Section 5. Section 6 is dedicated to the discussion of some implementation details. In Section 7, optimization through delta messages is discussed. We conclude the paper with a brief discussion of our results.

2 Motivation

In this section, we use an example of a time warp execution that consists of a process implementing a *set* object. In particular, we demonstrate how the type-specific properties such as *commutativity* [Kor83, SS84, Wei89] and *invalidated-by* [HW91] relations on operations on a set object can be used to reduce rollback in the system. A set object supports three types of operations: *insert*, *delete*, and *member*. Two insert operations *commute*, since the ordering of elements in a set is irrelevant. Similarly, an insert operation commutes with a delete operation, if the two operations operate on *different* elements. The same holds for an insert or delete operation with respect to a member operation. We say that an operation *invalidates* another if the insertion of the first operation before the second in an execution will make the response generated by the second operation invalid. A delete operation which gives a *no* response is invalidated by an insert operation, since the insert operation may insert the element to be deleted into the set, making the earlier response *no* invalid. The correct response would then be *yes*. However, a delete operation with a *yes* response is not invalidated by an insert operation.

Input Queue:	$ins(y)@10$	$ins(x)@14$	$mem(x)@20$	$del(x)@25$	$del(z)@32$	...
		$\uparrow$	$LVT = 14$			
Output Queue:	$ok@10$	$ok@14$				
State Queue:	$\{y\} \oplus \phi@10$	$\{x, y\} \oplus \phi@14$				

Figure 1: A time warp execution on the set object S

In Figure 1, we show the state of a partial time warp execution on a set object S. The implementation of the set object in the TWM comprises of a local state queue, an input queue, and an output queue. Both input queue and output queue hold messages. Each message m in the queue comprises of six fields: the *send virtual time (SVT)* of m, the *receive virtual time (RVT)* of m, the *sender* of m, the *receiver* of m, the type of m (positive or negative)[2], and finally the content of the message m. The state queue is a list of checkpointed states and has three fields per entry: the *local virtual time LVT* of the specific state, the *state* itself, and an *intention list*. An object is analogous to a server in the client/server model. It accepts operation invocations from the input queue in the RVT order, performs the operations (setting the LVT to the RVT of the invocation) and returns the responses to the caller via a message. The intention list corresponding to the LVT, which was not part of the original TWM, will hold all the deferred updates that must be applied to yield the correct state, for that value of the LVT. For simplicity, the figures in this paper only depict the fields that are relevant to our discussion. The condensed form of an input message is $op\_inv@RVT$ and of an output message is $resp@SVT$. An entry of the state queue is $state \oplus intention@LVT$.

Consider that in Figure 1, an out-of-order message to insert an element x at virtual time 4 (denoted $ins(x)@4$) arrives at S. In the original TWM, this would necessitate a rollback of S, causing its LVT to be reset to 0 which is smaller than the RVT of the new $ins(x)$ message. However, since $ins(x)@4$ commutes with $ins(y)@10$, we can allow $ins(x)@4$ to be executed after $ins(y)@10$. The resulting state as well as the response would be the same as if the two operations were executed in the prescribed order. Furthermore, since $ins(x)@4$ commutes with $ins(x)@14$, $ins(x)@4$ can be executed after virtual time 14 instead of at 4. The resulting state as well as the response is identical irrespective of whether $ins(x)@4$ is executed at virtual time 4 or after 14. Hence, rollback due to $ins(x)@4$ can be avoided for this particular case.

Input Queue:	$ins(x)@4$	$ins(y)@10$	$ins(x)@14$	$mem(x)@20$	$del(x)@25$	$del(z)@32$	...
			$\uparrow$	$LVT = 14$			
Output Queue:	$ok@4$	$ok@10$	$ok@14$				
State Queue:	$\{\} \oplus ins(x)@4$	$\{y\} \oplus ins(x)@10$	$\{x, y\} \oplus \phi@14$				

Figure 2: Processing $ins(x)@4$ in Figure 1

The resultant state of S after executing $ins(x)@4$ described above is illustrated in Figure 2. Note, however, the intermediate states (at virtual time 4 and 10) are no longer correct. Since the intermediate states are used only to implement rollback of the object from those specific values of LVT, we can recreate the correct intermediate states if and when it is necessary, by applying the intention list to the state. Thus, although the SVT of the response of $ins(x)@4$ is set to 4, the correct response as well as the correct state $\{x, y\}$ is generated by executing $ins(x)@4$ after virtual time 14 instead of at the intended virtual time 4.

[2]See [Jef85] for a discussion of regular (positive) and anti (negative) messages, which are analogous to the concept of matter and anti-matter in particle physics.

Next, we give an example to illustrate that commutativity of operation is not always applicable to avoid rollback. In this case, we apply an alternative notion, i.e., the invalidated-by relation, to handle rollback. Consider the state of the set object S in Figure 1, but with $LVT = 25$. Now suppose an out-of-order message arrives at S to insert x at virtual time 15, i.e., $ins(x)$@15. Note that $ins(x)$@15 does not commute with $mem(x)$@20 and $del(x)$@25 since a different order of executing these operations may result in different object states, or different response. However, we can observe that $ins(x)$@15 will not invalidate the affirmative responses of $mem(x)$@20 and $del(x)$@25. As a result, the output messages need not be withdrawn. We only need to roll the state queue back to virtual time 14, execute $ins(x)$@15 and re-execute $mem(x)$@20 and $del(x)$@25 to arrive at the correct state at virtual time 25, as shown in Figure 3. The reason we need to re-execute the operation is that unlike the commutativity relation, the invalidated-by relation does not guarantee the final state to be identical when the operations are executed in a different order. For example, if $ins(x)$@15 is executed after virtual time 25, x will be a member of S. On the other hand, if $ins(x)$@15 is executed at virtual time 15, x is not longer in S after the execution of $del(x)$@25.

Input Queue:	$ins(y)$@10	$ins(x)$@14	$ins(x)$@15	$mem(x)$@20	$del(x)$@25	$del(z)$@32	...
					$\uparrow$	$LVT = 25$	
Output Queue:	ok@10	ok@14	ok@15	yes@20	yes@25		
State Queue:	$\{y\} \oplus \phi$@10	$\{x, y\} \oplus \phi$@14	$\{x, y\} \oplus \phi$@15	$\{x, y\} \oplus \phi$@20	$\{y\} \oplus \phi$@25		

Figure 3: Processing $ins(x)$@15 in Figure 1 with LVT 25

Based on the above examples, we now give an informal overview of the protocol. In particular, we can use both the commutativity as well as the invalidated-by relations to minimize rollback in the system. Consider, for example, that in the state of Figure 1 when the LVT is already 25, an out-of-order message $ins(x)$@4 arrives. From our discussion based on Figures 2 and 3, we can execute $ins(x)$@4 after the state of S at time 14, since $ins(x)$@4 commutes with $ins(y)$@10 and $ins(x)$@14. Furthermore, since $ins(x)$@15 does not invalidate $mem(x)$@20 with response yes and $del(x)$@25 with response yes, we do not need to rollback or annihilate output messages. Instead the operation needs to be re-executed after time 14 to recreate the correct final state. We employ the commutativity relation before the invalidated-by relation, since only the commutativity relation can guarantee an identical object state on which the responses for the other operations are generated.

We conclude this section with a brief discussion for handling anti-messages that result when output messages are rolled back by TWM processes. For example, an anti-message for $ins(y)$@10 may arrive at the set object S in Figure 1. Since $ins(y)$ commutes with all subsequent messages (i.e., $ins(x)$@14, $mem(x)$@20, $del(x)$@25), the anti-message of $ins(y)$ can be executed after virtual time 25, instead of rolling S back to virtual time 10. The result is shown in Figure 4. As will be explained later, we develop refinements of the invalidated-by relation into an invalidate-on-insert relation for regular messages and an invalidate-on-delete relation for anti-messages.

Input Queue:	$ins(x)@14$	$mem(x)@20$	$del(x)@25$	$del(z)@32$	...
			$\uparrow$	$LVT = 25$	
Output Queue:	$ok@14$	$ok@20$	$yes@25$		
State Queue:	$\{x, y\} \oplus ins^{-}(y)@14$	$\{x, y\} \oplus ins^{-}(y)@20$	$\{\} \oplus \phi@25$		

Figure 4: Processing an anti-message $ins(y)@10$ when LVT is 25

Although the use of commutative messages is intuitive, it does not lead to optimal performance in reducing rollback. In this paper, we are using a *weaker* notion of commutativity and show that it can be adopted to further reduce rollback. Based on this weaker notion of commutativity, the message $ins(x)@4$ commutes with the first three operations in the above example (i.e., $ins(y)@10$, $ins(x)@14$ and $mem(x)@20$ with yes response). As a consequence, re-execution can start after the state at virtual time 20 instead of virtual time 15.

3 The Model

In this section, we will present the TWM model and define the relations, which capture semantics of operations, used in our protocol. The time warp system is composed of a set of entities distributed over the network. They communicate by sending messages to one another. Some of the entities provide a structured interface when they interact with other entities. We use the term *objects* to refer to such entities. Other entities that do not provide a structured interface are called *processes*. Objects are derived from abstract data type specifications and provide a set of operation types. Each operation type defines a template, from which operation invocations (calls to the object to perform some functions) can be made, by filling in the parameters. The objects receive operation invocations in the form of input messages and return the corresponding responses (results of the function calls) via output messages. All messages are processed in the RVT order. There are two kinds of messages in the TWM: regular messages and anti-messages. An anti-message is an exact copy of a regular message, but with a negative polarity. Its sole purpose is to annihilate the corresponding regular message with a positive polarity. The output queue of an entity consists of anti-messages to facilitate the annihilation of the corresponding regular message. For our implementation of objects, we use the inverse operation to cancel out the effect of an operation. In case that the inverse of an operation is undefined, a procedure may be attached to undo the effect of the operation in certain object states. This procedure may be just a partial function and hence not an inverse. If it cannot undo the operation in a particular object state, other technique such as before-image restoration, compensating operation (analogous to compensating transaction), or a rollback is needed. The anti-message for the operation $del(x)$ is denoted $del^{-}(x)$ and is expected to invoke an inverse operation of $del(x)$ if the operation has been executed. A message m for an object may contain either an operation invocation (inv), a response ($resp$) or general information. The input queue is ordered according to the RVT and the output queue is ordered according to the SVT of messages.

After the execution of an input message, there is a corresponding entry created in the state queue containing the resulting state of the object.

We distinguish between the terms operation invocation, operation response, and operation. We define an *operation* to be a *paired* operation *invocation* and the corresponding *response*. Thus an operation p has two parts: $\langle inv(p), resp(p)\rangle$. The same invocation $mem(x)$ corresponds to one of the two possible operations: $\langle mem(x), yes\rangle$ and $\langle mem(x), no\rangle$, depending on whether the current object state contains the element x or not. For the sake of simplicity, we represent an object state in terms of an operation sequence. Suppose an operation invocation $inv(p)$ is received by the object. Based on the current state, it generates a response $resp(p)$ and transits to another state. It is clear that the current state together with the operation $p = \langle inv(p), resp(p)\rangle$ uniquely determines the new state. We can denote an object state by a sequence of operations that have been executed on the object, starting from the initial state. For example, the state of the set object S in Figure 1 at virtual time 14 can be represented as $\langle ins(y), ok\rangle \cdot \langle ins(x), ok\rangle$[3], which is equivalent to $\{x, y\}$ for the initial state $\{\}$.

The behavior of an object can be represented by a set of *valid* operation sequences. Recall that an operation sequence represents an object state, as well as a history of how the object reaches that state. We say that an operation sequence is *valid* if and only if the response generated at each step of the sequence is consistent with the object behavior. For example, with respect to the initial state $\{\}$, $\langle ins(x), ok\rangle \cdot \langle mem(x), yes\rangle$ is a valid sequence for the set object but $\langle ins(x), ok\rangle \cdot \langle mem(x), no\rangle$ is not. Two operation sequences g and h are said to be *equivalent* if they cannot be distinguished in the future. For example, $\langle ins(x), ok\rangle \cdot \langle del(x), ok\rangle$ and $\langle mem(x), no\rangle$ are equivalent. More formally, g and h are equivalent if and only if for all sequences s, $g \cdot s$ is valid implies $h \cdot s$ is valid and vice versa. Equivalence of states is a weaker notion of being the "same" state. The semantics of two operations can be captured by the effect on the object when these two operations are executed by the object in different orders. An example is the notion of commutativity between two operations. In particular, we are using the *right-backward commute* relation [Wei89] and the *invalidated-by* relation [HW91].

Definition 1 *An operation q is said to* **right commute backward** *with another operation p whenever for all sequences h, if $h \cdot p \cdot q$ is valid then $h \cdot q \cdot p$ is valid and equivalent to $h \cdot p \cdot q$.*

Intuitively, we say q right commutes backward with p if whenever q can be executed on the state after p has been executed, it is *always* possible to move q back over p and execute them in the reverse order, such that the responses generated by the two operation invocations are the same and the resulting object states are equivalent.

The right-backward commute relation for the set object is illustrated in Table 1. An entry $\surd$ indicates that the operation in the first column (with parameter y) right commutes backward with the operation in the first row (with parameter x). In addition, a conditional entry indicates that the operation in the first column right commutes backward with the operation in the first row if

[3]The operator "$\cdot$" denotes concatenation of operations and operation sequences.

Set	$\langle ins(x), ok\rangle$	$\langle del(x), yes\rangle$	$\langle del(x), no\rangle$	$\langle mem(x), yes\rangle$	$\langle mem(x), no\rangle$
$\langle ins(y), ok\rangle$	$\surd$	$x \neq y$	$x \neq y$	$\surd$	$x \neq y$
$\langle del(y), yes\rangle$	$x \neq y$	$x \neq y$	$\surd$	$x \neq y$	$\surd$
$\langle del(y), no\rangle$	$\surd$	$x \neq y$	$\surd$	$\surd$	$\surd$
$\langle mem(y), yes\rangle$	$x \neq y$	$\surd$	$\surd$	$\surd$	$\surd$
$\langle mem(y), no\rangle$	$\surd$	$x \neq y$	$\surd$	$\surd$	$\surd$

Table 1: Right-backward commute relation for the set object

the stated condition holds. For example, the operation $\langle ins(x), ok\rangle$ right commutes backward with $\langle mem(x), yes\rangle$ since whenever it is possible to execute $\langle mem(x), yes\rangle$ and $\langle ins(x), ok\rangle$ in that order, we can swap them and execute $\langle ins(x), ok\rangle$ before $\langle mem(x), yes\rangle$. However, it must be pointed out that the right-backward commute relation may not be a symmetric relation. This makes the handling of regular messages and anti-messages slightly different, as described in the following section. In particular, $\langle mem(x), yes\rangle$ does not right commute backward with $\langle ins(x), ok\rangle$, since the original object state before the two operations may not contain the element x.

We now formally define the *invalidated-by* relation. Note that in the TWM, due to the distinction between regular messages and anti-messages, this relation needs to be refined for the two cases when an operation is inserted in a sequence as well as when it is withdrawn (annihilated). We term these two relations as *invalidated-on-insert* for regular messages and *invalidated-on-delete* for anti-messages.

Definition 2 *An operation p is said to* **invalidate-on-insert** *another operation q if there exist sequences g and h, such that $h \cdot p \cdot g$ is valid and $h \cdot g \cdot q$ is valid but $h \cdot p \cdot g \cdot q$ is invalid.*

Intuitively, we say that p invalidates q, if the insertion of p in the correct position in the execution can render the response generated by q invalid in at least one possible state. We include the condition that the sequence $h \cdot p \cdot g$ should be valid to rule out the possibility that p does not really invalidate q, but merely because it has already invalidated some operations in the subsequence g to make $h \cdot p \cdot g \cdot q$ invalid.

Set	$\langle ins(x), ok\rangle$	$\langle del(x), yes\rangle$	$\langle del(x), no\rangle$	$\langle mem(x), yes\rangle$	$\langle mem(x), no\rangle$
$\langle ins(y), ok\rangle$					
$\langle del(y), yes\rangle$		$x = y$			
$\langle del(y), no\rangle$	$x = y$				
$\langle mem(y), yes\rangle$		$x = y$			
$\langle mem(y), no\rangle$	$x = y$				

Table 2: Invalidated-on-insert relation for the set object

The invalidated-on-insert relation for the set object is depicted in Table 2. A conditional entry implies possible invalidation of the operation in the first column (parameter y) by the operation in the first row (parameter x) when the condition is satisfied. An empty entry indicates no invalidation.

For example, the insertion of an $ins(x)$ operation in the set object may make the response *no* generated to a $mem(x)$ operation invalid. We say that $\langle ins(x), ok \rangle$ invalidates-on-insert $\langle mem(x), no \rangle$. However, the same operation cannot make the response *yes* to a $mem(x)$ operation invalid.

Definition 3 *An operation p is said to* **invalidate-on-delete** *another operation q if there exist sequences g and h, such that $h \cdot p \cdot g \cdot q$ is valid and $h \cdot g$ is valid but $h \cdot g \cdot q$ is invalid.*

Intuitively, this relation corresponds to the withdrawal of an operation p from a sequence and requires that the responses generated by operations already executed after p remain valid. This relation captures the semantics required on the processing of an anti-message, which dictates the annihilation of an operation. We have to assert the sequence $h \cdot g$ to be valid, to rule out the possibility that $h \cdot g \cdot q$ is invalid no matter what q is, simply because $h \cdot g$ is invalid.

Set	$\langle ins(x), ok \rangle$	$\langle del(x), yes \rangle$	$\langle del(x), no \rangle$	$\langle mem(x), yes \rangle$	$\langle mem(x), no \rangle$
$\langle ins(y), ok \rangle$					
$\langle del(y), yes \rangle$	$x = y$				
$\langle del(y), no \rangle$		$x = y$			
$\langle mem(y), yes \rangle$	$x = y$				
$\langle mem(y), no \rangle$		$x = y$			

Table 3: Invalidated-on-delete relation for the set object

The invalidated-on-delete relation is shown in Table 3. The notation used parallels the invalidated-on-insert relation. In this case, however, the withdrawal of an $ins(x)$ operation in the set object does not make the response *no* generated to a $mem(x)$ operation invocation invalid, but it may make the response *yes* invalid, since the insertion of x is withdrawn. Thus $\langle ins(x), ok \rangle$ does not invalidate-on-delete $\langle mem(x), no \rangle$, but invalidates-on-delete $\langle mem(x), yes \rangle$.

The relations between operations defined above are based on *paired* operation invocation and response. When an out-of-order operation invocation is received, we do not know the response without executing it. This creates a predicament of determining the response of an invocation to apply the relations based on the paired invocation and response. One way out of this predicament is to perform a trial execution and use the response generated to form the paired invocation and response. Another way is to be conservative: assuming that the response may be of any possible form. This latter approach results in a collapse of the entries in the tables. An invocation $inv(q)$ right commutes backward with operation p only when every entry of the form $\langle inv(q), \_ \rangle$ right commutes backward with p. For example, now $del(y)$ right commutes backward with $\langle ins(x), ok \rangle$ when $x \neq y$. This results in a *restricted* form of the tables for the relations. Note that the problem of determining the response of an operation invocation is limited to the commutativity relation. In contrast, when we employ the invalidated-by relation, the operation invocation has already been executed and its response is known. Hence the two invalidated-by relations can be used in their original form.

4 Semantics-based Time Warp Protocol

In this section, we develop a protocol that is an extension of the original TWM. The main modification is in the processing of input messages to minimize rollback, as described in Section 2. In the original TWM, an input message arriving at an entity with a larger RVT than the LVT of the entity can simply be inserted in the input queue. Similarly, an anti-message corresponding to an unprocessed message in the input queue annihilates its positive counterpart, leaving no trace of the message. On the other hand, if an anti-message with an earlier RVT than the LVT arrives before its positive counterpart, the anti-message is inserted in the input queue and waits for its positive counterpart to arrive and annihilate. Thus, the modification in our proposed protocol is to handle the case when the arrival of an out-of-order message or an out-of-order anti-message causes a rollback in the original TWM.

We now give a brief overview to illustrate how an input message to an object ($inv(p)$ or $inv^{-}(p)$) is handled in our protocol. We denote the sequence of operation invocations (input messages) that have been processed with a RVT greater than the RVT of the new message as α. The main task of the protocol is to partition α into $\alpha_1 \cdot \alpha_2 \cdot \alpha_3 \cdot \alpha_4$, satisfying certain properties. First, we identify the prefix α_1 such that operations in α_1 and $inv(p)$ have a commutative relationship. We then execute $inv(p)$ in the state after α_1 and appropriately update the intention lists of intermediate states corresponding to α_1. Now the response to $inv(p)$ is known, and hence operation p is known. We next identify α_2 in which every operation q is not invalidated by p. Operations in α_2 are re-executed to recreate correct object states without the concern that the responses of the operations in α_2 will be invalidated. That is, the re-execution does not result in cascading rollback. After the sequence α_2, we use the rollback technique of the original TWM to handle the sequence α_3 which necessitates both local re-execution as well as withdrawal of output messages. Finally, we identify the sequence α_4 which is idempotent with respect to p, such that the state at the beginning of α_4 is the same whether p has been executed or not. The algorithm terminates as soon as α_4 is reached. Thus, rollback is necessary only for α_2 and α_3, with α_2 causing local rollback whereas α_3 causing global or cascading rollback.

We use the commutativity relation and the invalidated-by relation to identify the sequences α_1 and α_2 with respect to the out-of-order message. However, this process depends on the message type, i.e., whether the message being handled is a regular message $inv(p)$ or an anti-message $inv^{-}(p)$. In particular, for the out-of-order message $inv(p)$, it must be the case that for all operations $q \in \alpha_1$, $inv(p)$ right commutes backward with q. On the other hand, to deal with the out-of-order anti-message $inv^{-}(p)$, it must be the case that for all operations $q \in \alpha_1$, q right commutes backward with $inv(p)$. For a regular message, p must not invalidate-on-insert any operation $q \in \alpha_2$ whereas for an anti-message, p must not invalidate-on-delete any operation $q \in \alpha_2$. The protocol is depicted in Figure 5.

```
When a message m is received at an object:
begin
    if the RVT of m > LVT then insert m into the input queue
    elseif m is regular and the anti-message for m is in the input queue then annihilate both messages
    elseif m is regular then /* now m will cause a rollback */
        rollback the LVT, input queue, output queue and state queue to the point before the RVT of m
        let m carry inv(p), create an entry for inv(p) in the state queue
        find α1, starting from the current LVT such that inv(p) right commutes backward with each q ∈ α1
        foreach q ∈ α1 do
            insert inv(p) into the intention list of q and advance LVT
        endfor
        reconstruct object state and apply inv(p) on the object to yield resp(p)
        now p = ⟨inv(p), resp(p)⟩
        find α2, starting from the current LVT such that p does not invalidate-on-insert each q ∈ α2
        foreach q ∈ α2 do
            re-execute q locally, without sending any message
            if new state is equivalent to the old one then return /* early termination: α4 */
        endfor
        foreach q in the remaining sequence of rolled back operations do
            re-execute q, send out anti-message for old response and send out message for new response
            if new state is equivalent to the old one then return /* early termination: α4 */
        endfor
    elseif there is not a matching regular message for m then insert m into input queue
    else /* this anti-message will cause a rollback */
        rollback the LVT, input queue, output queue and state queue to the point before the RVT of m
        let m carry inv⁻(p)
        find α1, starting from the current LVT such that each q ∈ α1 right commutes backward with inv(p)
        foreach q ∈ α1 do
            insert inv⁻(p) into the intention list of q and advance LVT
        endfor
        perform the inverse operation on the state of the last q
        let p = ⟨inv(p), resp(p)⟩
        find α2, starting from the current LVT such that p does not invalidate-on-delete each q ∈ α2
        foreach q ∈ α2 do
            re-execute q locally, without sending any message
            if new state is equivalent to the old one then return /* early termination: α4 */
        endfor
        foreach q in the remaining sequence of rolled back operations do
            re-execute q, send out anti-message for old response and send out message for new response
            if new state is equivalent to the old one then return /* early termination: α4 */
        endfor
    endif
end
```

Figure 5: Semantics-based time warp protocol

5 Integrating Right Backward Commutativity with Forward Commutativity

When operation responses are taken into account, our protocol can be improved through the introduction of the *forward commute relation* [Wei89]. By combinating the forward commute relation with the right-backward commute relation, the amount of rollback can further be reduced. We first give a formal definition of forward commutativity.

Definition 4 *An operation p is said to* **commute forward** *with another operation q whenever for any sequence h, if $h \cdot p$ and $h \cdot q$ are valid then $h \cdot p \cdot q$ and $h \cdot q \cdot p$ are both valid and are equivalent.*

Intuitively, we say p commutes forward with q if whenever both p and q can be executed on the same state, it is *always* possible to execute p followed by q and to execute q followed by p on that state such that the responses generated by the operation invocations are the same and the resulting object states are equivalent, independent of the order of execution of the two operations. Forward commutativity is a symmetric relation, i.e. p commutes forward with q if and only if q commutes forward with p. Forward commutativity closely resembles the usual sense of commutativity.

Set	$\langle ins(x), ok\rangle$	$\langle del(x), yes\rangle$	$\langle del(x), no\rangle$	$\langle mem(x), yes\rangle$	$\langle mem(x), no\rangle$
$\langle ins(y), ok\rangle$	$\surd$	$x \neq y$	$x \neq y$	$\surd$	$x \neq y$
$\langle del(y), yes\rangle$	$x \neq y$	$x \neq y$	$\surd$	$x \neq y$	$\surd$
$\langle del(y), no\rangle$	$x \neq y$	$\surd$	$\surd$	$\surd$	$\surd$
$\langle mem(y), yes\rangle$	$\surd$	$x \neq y$	$\surd$	$\surd$	$\surd$
$\langle mem(y), no\rangle$	$x \neq y$	$\surd$	$\surd$	$\surd$	$\surd$

Table 4: Forward commute relation for the set object

The forward commute relation for the set object is depicted in Table 4. The entry $\surd$ indicates that the two operations commute forward with each other and a conditional entry indicates commutativity only under the specified condition. For example, the operation $\langle ins(x), ok\rangle$ commutes forward with $\langle ins(y), ok\rangle$ but $\langle ins(x), ok\rangle$ only commutes forward with $\langle del(y), yes\rangle$ when the two elements are distinct.

5.1 Processing an Out-of-order Message

We can refine the definition for the sequence α_1 and the new definition is weaker than the old one, effectively elongating the sequence. The same protocol in Figure 5 can be used, but with the new definition for α_1. Let $\mathcal{FC}$ and $\mathcal{RBC}$ denote the forward and right-backward commute relations. We say that $(q, p) \in \mathcal{RBC}$ if q right commutes backward with p. Let the operation sequence from the beginning up to α_1 be denoted as α_0.

When an out-of-order invocation $inv(p)$ is received, we define formally α_1 to be the maximal sequence that satisfies the following:

$\exists n, f_1, b_1, ..., f_n, b_n$ such that $\alpha_1 = f_1 \cdot b_1 \cdot ... \cdot f_n \cdot b_n \wedge \forall i \in [1..n], \forall q \in f_i, (p, q) \in \mathcal{FC} \wedge \forall q \in b_i, (p, q) \in \mathcal{RBC} \wedge valid(\alpha_0 \cdot f_1 \cdot b_1 \cdot ... \cdot f_i \cdot b_i)$.

Intuitively, α_1 is composed of alternating subsequences f_i and b_i such that p commutes forward with all operations in f_i and right commutes backward with all operations in b_i. If we decompose α_1 into $f_1 \cdot b_1 \cdot ... \cdot f_n \cdot b_n$, we can avoid any rollback, since the operation p can always be executed at the end of α_1 and the resultant operation sequence remains valid and equivalent to the original one. An additional constraint is that the operation p must be executable at any junction between a backward subsequence b_i and the next forward subsequence f_{i+1} (expressed as the clause $valid(\alpha_0 \cdot f_1 \cdot b_1 \cdot \ldots \cdot f_i \cdot b_i)$ in the condition). This verification of responses must be performed in the extended protocol, since it is essential in order to guarantee the correctness of the protocol. If we drop the forward commute relation, it degenerates into the protocol in Figure 5, in which $\alpha_1 = b_1$. A formal proof of correctness can be found in the Appendix.

5.2 Processing an Anti-message

To cater for anti-messages with the extended notion, we need to define the forward commute relation on inverse operations as well. Mathematically, we require the inverse operation p^- to satisfy the *identity property*:

$$\forall h, valid(h \cdot p \cdot p^-) \wedge equiv(h \cdot p \cdot p^-, h).$$

The forward commute relation definition can also cover inverse operations without any modification. In Table 5, the relation for the set object in the context of inverse operation is depicted, which is a generalization over Table 4.

Set	$\langle ins(x), ok\rangle$	...	$\langle ins^-(x), ok\rangle$	$\langle del^-(x), yes\rangle$	$\langle del^-(x), no\rangle$	$\langle mem^-(x), yes\rangle$	$\langle mem^-(x), no\rangle$
$\langle ins(y), ok\rangle$	$\surd$	...	$x \neq y$	$\surd$	$\surd$	$\surd$	$\surd$
$\langle del(y), yes\rangle$	$x \neq y$	...	$x \neq y$	$x \neq y$	$\surd$	$\surd$	$\surd$
$\langle del(y), no\rangle$	$x \neq y$	...	$x \neq y$	$x \neq y$	$\surd$	$\surd$	$\surd$
$\langle mem(y), yes\rangle$	$\surd$	...	$x \neq y$	$\surd$	$\surd$	$\surd$	$\surd$
$\langle mem(y), no\rangle$	$x \neq y$	...	$\surd$	$x \neq y$	$\surd$	$\surd$	$\surd$
$\langle ins^-(y), ok\rangle$	$\surd$	...	$x \neq y$	$x \neq y$	$\surd$	$\surd$	$\surd$
...	...	...	...	...	...	...	...

Table 5: Forward commute relation for the set object (including inverse operation)

Inverse operations do not have responses, since they are to eliminate the effect of the underlying operation. However, the response of the operation gives us more information to define the inverse. For example, the inverse to $\langle del(x), yes\rangle$ is $ins(x)$ but the inverse to $\langle del(x), no\rangle$ is **id** (the identity function). Therefore, we still use the notation of $\langle del^-(x), yes\rangle$ to represent the inverse operation to the operation $\langle del(x), yes\rangle$, bearing in mind that yes is not a response of $del^-(x)$. Now we see that $\langle del^-(x), yes\rangle$ commutes forward with $\langle ins(y), ok\rangle$, since $\langle ins(x), ok\rangle$ commutes forward with $\langle ins(y), ok\rangle$. Similarly, $\langle del^-(x), yes\rangle$ commutes forward with $\langle del(y), yes\rangle$ only when $x \neq y$. Note that the lower left quarter of the table is the same as the upper right quarter, due to symmetry and is therefore not shown. The lower right quarter involves forward commutativity between two inverse operations, which is not used in the protocol.

Formally, α_1 can be characterized as the maximal sequence that satisfies

$$\exists f, b, \alpha_1 = b \cdot f \wedge \forall q \in b, (q, p) \in \mathcal{RBC} \wedge \forall q \in f, (q, p^-) \in \mathcal{FC}.$$

To withdraw the effect of p, we push every operation q in b over p so that p appears at the end of b using the right-backward commutativity. We can then withdraw the effect of p after b by performing the inverse operation p^-. Now we push forward this p^- through f, using the forward commutativity property and finally, have the effect of p withdrawn after f. In other words,

$\alpha_0 \cdot p \cdot p^{-} \cdot \alpha_1$ is equivalent to $\alpha_0 \cdot p \cdot \alpha_1 \cdot p^{-}$. Similarly, if we drop the commutativity for the inverse operation, the protocol reduces to the protocol in Figure 5 with $\alpha_1 = b$.

6 Handling Multiple Out-of-order and Anti-messages

In contrast to the flat structure of the rollback processing of TWM, the rollback scheme in our protocol is *nested*. In particular, if a new out-of-order message arrives during the rollback processing of another out-of-order message, we cannot switch to the new message and be sure that the rollback processing of the new message will include that of the current message. We want to preserve both processing. In our scheme, the new message can be viewed as an interrupt that suspends the rollback processing of the current message. After the interrupt (rollback processing) is completed for the new message, the suspended rollback processing of the earlier message resumes towards completion.

A message is said to be *outstanding* if it is being processed but the rollback processing has not yet completed. A global list $Outstand\_msg$ is defined for outstanding messages, initialized to an empty list. When the rollback processing of an outstanding message has completed, it is removed from the list. This list is ordered according to the RVT of the messages and each message has an additional field called *phase* which indicates which of the α_i sequence the processing of the message is in. The variable $Curr\_time$ indicates the termination time of the protocol. The variable $Proto\_phase$ monitors the current phase of the protocol. Stated in another way, $Proto\_phase$ is the most advanced value of all the phases of messages in $Outstand\_msg$. We use the notation $m.field$ to refer to the fields of message m, or fields of the entry in the output queue whose $LVT = m.RVT$. Let $prev(m)$ denote the message m' just before m in the input queue and $next(m)$ denote the one just after.

The protocol processes the set of outstanding messages in a "lock-step" manner. The term lock-step is used to mean that whenever the LVT is advanced to the RVT of the next operation, the effect of every eligible outstanding message will be considered. Let us call an outstanding message with a smaller RVT than the LVT a *mature* message. Only mature messages may cause a rollback for that LVT. If a new message has a RVT greater than the LVT, it is inserted into $Outstand\_msg$, waiting to be processed when the LVT advances[4]. If its RVT is smaller than the LVT, the current execution is interrupted and a new copy of the protocol is invoked. The new copy will run until its LVT is at least the LVT of the interrupted copy. At this moment, this new copy terminates. Only the first invocation of the protocol, caused by the first out-of-order message, can run to completion to the virtual time just before the rollback (the time $Curr\_time$). All other invocations will terminate when their LVT's catch up with the LVT of the invocation they interrupt. The protocol starts by taking the first operation after the rolled back virtual time

[4]Note that if the RVT of the message is already greater than the LVT when it was *before* the arrival of the first out-of-order message (this LVT value is stored in $Curr\_time$), the message is simply inserted into the input queue and will *not* cause any rollback. Therefore in this case, sooner or later the LVT will be greater than its RVT and the message will be handled.

When a message m is received:
begin
    disable interrupt
    if simple cases **then** perform the appropriate actions, enable interrupt, and **return**
    if $Outstand\_msg = \phi$ **then** /* first invocation */
        $Curr\_time \leftarrow LVT$
        $m.phase \leftarrow \alpha_1$ and insert m into $Outstand\_msg$
        $LVT \leftarrow m.RVT$
        enable interrupt
        call $execute(m, Curr\_time)$
        $Outstand\_msg \leftarrow \phi$
    else /* subsequent invocation */
        $m.phase \leftarrow \alpha_1$ and insert m into $Outstand\_msg$
        enable interrupt
        call $execute(m, LVT)$
    endif
end

Figure 6: Nested Semantics-based Time Warp Protocol

(the first operation in α) and setting the LVT to its RVT. This operation is checked with all mature messages for commutativity and non-invalidation, then the appropriate actions are taken. The next operation is then considered and the LVT advanced. In this way, for every operation q in α, when q is being considered, all operations that happen before q are known and handled (or will be known and will be processed). The processing of operations in α continues until $Curr\_time$ is reached or early termination is detected. The protocol is presented in Figures 6 and 7.

When an out-of-order message that may cause a rollback is encountered, a new invocation of the procedure is made. An entry is allocated for it in the state queue and is properly initialized (see line (A) in Figure 7). The protocol then starts to execute in phase α_1. For each operation q, the condition for commutativity is checked for each mature message and when at least *one* of them fails to commute with q, the protocol advances to the next phase α_2. It is because that if one of them fails to commute, all the states after that offending mature message will no longer be guaranteed to be equivalent. We can only reconstruct the object state and starts re-execution of all other operations in α. When the commutativity condition is being verified, the mature message is also inserted into the intention list for the state of q. To start phase α_2, the current object state is reconstructed using the intention list for the state of the last operation that is still commutative. Note that in the procedure presented, the out-of-order message being processed is already in the intention list when phase α_1 terminates (line (B)). As a result, the out-of-order message will have been incorporated, by applying the intention list on that last state. The non-invalidation condition is then verified for each operation q in the rest of α. The protocol enters phase α_3 if q is invalidated by at least one of the mature messages. In phase α_2, each q is also re-executed, but without sending out any message or anti-message. We also check for early termination of the protocol (identifying α_4, see line (C)). In phase α_3, operations are re-executed; output messages are regenerated; and

```
procedure execute(msg, T) /* execute message msg up to virtual time T */
begin
    create an entry for msg in the state queue with timestamp msg.RVT
    msg.state ← prev(msg).state and add msg to msg.int_list                      .....(A)
    q ← next(msg), LVT ← q.RVT, Proto_phase ← α1
α1 : if LVT > T then return /* normal termination */
    foreach m ∈ Outstand_msg with m.RVT ≤ LVT do
        check for commutativity, put m into q.int_list, update m.phase, and change Proto_phase to α2 if needed
    if Proto_phase = α1 then q ← next(q), advance LVT, and goto α1
     /* bring the object state up-to-date for re-execution in α2 */
    apply prev(q).int_list to prev(q).state and clear the intention list          .....(B)
α2: if LVT > T then return /* normal termination */
    foreach m ∈ Outstand_msg with m.RVT ≤ LVT do
        check for non-invalidation, update m.phase, and change Proto_phase to α3 if needed
    if Proto_phase = α3 then goto α3
    apply operation carried by q on prev(q).state to get q.state
    if states are equivalent then remove all m whose RVT ≤ LVT from Outstand_msg and return /* early term. */   (C)
    q ← next(q), advance LVT, and goto α2
α3: if LVT > T then return /* normal termination */
    foreach m ∈ Outstand_msg with m.RVT ≤ LVT do remove m from Outstand_msg     .....(D)
    apply operation carried by q on prev(q).state to get q.state
    withdraw old response and send new response
    if states are equivalent then remove all m whose RVT ≤ LVT from Outstand_msg and return /* early termination */
    q ← next(q) and advance LVT
    goto α3
end
```

Figure 7: The Procedure *execute*

anti-messages are sent. Early termination is also monitored. Once the protocol enters phase α_3, all effects of mature messages have been incorporated and they can therefore be discarded from *Outstand_msg* (see line (D)).

7 Integrating Anti-message with Subsequent Regular Message

In this section, we introduce a new type of message, the *delta-message*, in addition to the two existing types of messages (the regular and anti-messages). We will illustrate how this new type of message can be used in the protocol.

7.1 Definition of Delta-messages

In lieu of rollbacks, quite often the re-execution of a process will only result in a slightly different behavior, namely, a slight change of the out-message content, with the same RVT. The original TWM calls for an anti-message, followed by a regular message to be sent. When we do not assume any underlying property of the message passing mechanism, the two messages may cause rollback at different times. If we assume both messages to be delivered in FIFO order, the performance may still degrade. The reason is that the conditions for avoiding rollback are different for regular messages and anti-messages. The computation must satisfy both conditions to avoid rollback. For example when an operation is to be changed from p_1 to p_2, we will send an anti-message for p_1 and then a regular message for p_2. To reduce rollback, we require α_1 and α_2 to satisfy the following criteria:

$\forall q \in \alpha_1, (q, p_1) \in \mathcal{RBC} \wedge (p_2, q) \in \mathcal{RBC}$ and

$\forall q \in \alpha_2, (q, p_1) \notin \mathcal{I}_{ins} \wedge (q, p_2) \notin \mathcal{I}_{del}$.

We can combine both conditions by exploiting the semantics of both messages in an integrated manner. Let $p^\Delta = [p_1 \rightarrow p_2]$ denote the combined *delta* operation for operations p_1 and p_2. We then only require that

$\forall q \in \alpha_2, (q, p^\Delta) \notin \mathcal{I}_{chg}$,

where $\mathcal{I}_{chg}$ is the invalidated-on-change relation, which parallels both invalidated-on-insert and invalidated-on-delete relations.

Definition 5 *A change of operation from p_1 to p_2 is said to* **invalidate-on-change** *another operation q if there exist sequences g and h, such that $h \cdot p_1 \cdot g \cdot q$ and $h \cdot p_2 \cdot g$ are valid but $h \cdot p_2 \cdot g \cdot q$ is not valid.*

To simplify our protocol, the change of operation from p_1 to p_2 is assumed to be only a change in the parameter of the operation, whereas the *name* of the operation remains the same. We call such message a *delta-message*. For a more general change of operation, we would model with the traditional approach: sending an anti-message for p_1 and then a regular message for p_2. We denote $p^\Delta = [p_1 \rightarrow p_2]$ by $\langle op^\Delta(\overline{x}_1 : \overline{x}_2), \overline{r}_1 : \overline{r}_2 \rangle$ when $p_1 = \langle op(\overline{x}_1), \overline{r}_1 \rangle$ and $p_2 = \langle op(\overline{x}_2), \overline{r}_2 \rangle$.

Set	$\langle ins^\Delta(x_1 : x_2), ok : ok \rangle$	$\langle del^\Delta(x_1 : x_2), yes : yes \rangle$	$\langle del^\Delta(x_1 : x_2), yes : no \rangle$	$\langle del^\Delta(x_1 : x_2), no : yes \rangle$	$\langle del^\Delta(x_1 : x_2), no : no \rangle$	$\langle mem^\Delta(x_1 : x_2), \_ : \_ \rangle$
$\langle ins(y), ok \rangle$						
$\langle del(y), yes \rangle$	$x_1 = y$	$x_2 = y$		$x_2 = y$		
$\langle del(y), no \rangle$	$x_2 = y$	$x_1 = y$	$x_1 = y$			
$\langle mem(y), yes \rangle$	$x_1 = y$	$x_2 = y$		$x_2 = y$		
$\langle mem(y), no \rangle$	$x_2 = y$	$x_1 = y$	$x_1 = y$			

Table 6: Invalidated-on-change relation for the set object

Queue	$\langle enq(x), ok \rangle$	$\langle deq(), x \rangle$	$\langle del(), \perp \rangle$
$\langle enq(y), ok \rangle$			
$\langle deq(), y \rangle$	$\times$	$\times$	
$\langle deq(), \perp \rangle$	$\times$		

Table 7: Invalidated-on-insert relation for the queue object

Consider the set object, a change in the element to be inserted from x_1 to x_2 is not going to invalidate the result of $\langle del(y), yes \rangle$ if $x_1 \neq y$. Similarly, it does not invalidate the result of $\langle del(y), no \rangle$ if $x_2 \neq y$. The complete invalidated-on-change relation is shown in Table 6. It is the

Queue	$\langle enq(x), ok\rangle$	$\langle deq(), x\rangle$	$\langle del(), \perp\rangle$
$\langle enq(y), ok\rangle$			
$\langle deq(), y\rangle$	×	×	
$\langle deq(), \perp\rangle$		×	

Table 8: Invalidated-on-delete relation for the queue object

Queue	$\langle enq^{\Delta}(x_1 : x_2), ok : ok\rangle$	$\langle deq^{\Delta}(:), x_1 : x_2\rangle$	$\langle deq^{\Delta}(:), x_1 : \perp\rangle$	$\langle deq^{\Delta}(:), \perp : x_2\rangle$	$\langle deq^{\Delta}(:), \perp : \perp\rangle$
$\langle enq(y), ok\rangle$					
$\langle deq(), y\rangle$	×				
$\langle deq(), \perp\rangle$					

Table 9: Invalidated-on-change relation for the queue object

union of the two invalidation relations and we gain nothing. In fact, it can be shown that the invalidated-on-change relation is never stronger than the union of the two invalidation relations. If we consider the queue object (Tables 7, 8 and 9), we obtain an amazing improvement: the invalidated-on-change relation is even weaker than the intersection of the two invalidation relations. With the invalidated-on-change relation, the length of α_2 can be greatly increased.

7.2 Processing a Delta-message

When a delta-message m^{Δ} is received, it is processed in a similar way to a regular message or an anti-message. We divide the sequence into four subsequences $\alpha = \alpha_1 \cdot \alpha_2 \cdot \alpha_3 \cdot \alpha_4$. Let m^{Δ} carry $p\Delta = [p_1 \rightarrow p_2]$. The first two sequences should satisfy

$\alpha_1 = \min(\alpha_1$ for anti-message $inv^{-}(p_1), \alpha_1$ for regular message $inv(p_2))$ and

$\forall q \in \alpha_2, (q, p^{\Delta}) \notin \mathcal{I}_{chg}$.

The characterization of α_3 and α_4 remains the same as before.

The original protocol proceeds by performing the inverse operation p_1^{-} at the end of α_1 and then the operation p_2. Alternatively, we can perform the delta operation there. This is very useful to save overhead. For example a change of parameter in an *enqueue* operation only results in a change in one of the queue elements, which can be done more efficiently than undoing the enqueue operation and then doing another enqueue. A delta message of $inc(5)$ to $inc(8)$ for a counter object can be expressed as $inc^{\Delta}(5:8)$ which is equivalent to $inc(3)$. All operations in α_2 are then re-executed to recreate valid object states, *without* sending out any messages. Operations in α_3 are rolled back and rolled forward, as before.

After delta-messages are defined, one may suggest that anti-delta-messages should exist to rollback the result of a delta-message. This turns out to be not necessary. As the semantics of a delta-message tells, it is a *change*. Withdrawing a delta-message only means the change should not be made. As a result, only the new operation p_2 needs to be withdrawn. The old operation p_1 had already been revoked when the delta message is processed. Only an anti-message for p_2 is necessary to cancel the remaining effect of the delta-message. At the receiver, if the delta-message has not yet been processed, the anti-message will annihilate only the changed portion p_2, leaving an anti-message for p_1 as a net result. This p_1^- will annihilate p_1 when p_1 finally arrives. This is not a complete annihilation (just like a proton annihilates with an electron to form a neutron). In practice, the sender only has a register of the message for p_2 in its output queue. The delta-message is generated as a result of the combination of the old anti-message and the new message.

Successive delta-messages may be composed if they are of the same virtual timestamp at the receiver. For example, the delta-messages $p_1^\Delta = [p_1 \rightarrow p_2]$ and $p_2^\Delta = [p_2 \rightarrow p_3]$ can be combined to form $p^\Delta = [p_1 \rightarrow p_3]$, if they have not yet been processed.

8 Discussion

In this paper, we have developed semantics-based time warp protocols for reducing rollback in distributed discrete event simulations. The underlying approach is to model processes in a distributed simulation as objects with abstract data type specifications. Since such objects can be accessed only via a set of well-defined operation types, we can use the type-specific properties (i.e., commutativity relation and invalidated-by relation) of these operations to reduce rollback in the system. We feel that this approach is feasible since objects such as queues, and sets are widely used in simulations. The performance can be improved even further for large objects and more expensive operations, such as the determinant operation on a matrix object. Furthermore, with a wider acceptance of object-oriented approach for programming, our approach would be particularly beneficial. Although our protocols place additional responsibility on the programmers, this additional burden only amounts to specifying the three relations as a part of the implementation of objects. However, since these relations are completely local to an object, program design can be made modular, extensible and reusable.

Our approach to the TWM imposes a nested rollback structure on the computation in contrast to the flat rollback structure of conventional TWM. The initial protocol has therefore been extended to cope with this situation. Furthermore, there are several optimizations that can be adopted in the proposed protocols. Recall that we have used a restricted form of the table for the forward commute and right-backward commute relations when identifying the sequence α_1. We can relax this constraint on an anti-message, since when an anti-message $inv^-(p)$ is received, $inv(p)$ has been executed and the response is known. The two commutativity relations can be used in their original form. Another optimization that we have made is the use of the notion of *delta* messages. A delta message is a combination of an anti-message and the subsequent regular message. With a delta message, we capture the notion of delta change in an output message sent when a process is rolled back and then rolled forward. We have developed the invalidated-on-change relation that incorporates the notion of delta messages. Through this new relation and combining the existing relations, we are able to further reduce rollback in the TWM model.

We are currently investigating other synchronization problems in message passing systems that can benefit from the approach of exploiting the semantics of messages. For example, the performance of crash recovery and checkpointing can be improved [LA93] and the ordering constraints in broadcast and multicast protocols can be relaxed, when we take into account the semantics of messages.

Acknowledgement

We would like to express our gratitude to Jonathan R. Agre for technical supports on discrete event simulation and time warp mechanism. He helps in bringing in relevant work and comparison and in proofreading parts of this paper.

References

[ABHN91] Mustaque Ahamad, James E. Burns, Phillip W. Hutto, and Gil Neiger. Causal memory. Technical report, Georgia Institute of Technology, College of Computing, Atlanta, GA30332-2593, 1991.

[BJKS88] K.P. Birman, T.A. Joseph, K. Kane, and F. Schmuck. ISIS - a distributed programming environment user's guide and reference manual. Technical report, Cornell University, Department of Computer Science, March 1988.

[BL92] R.L. Bagrodia and W.T. Liao. Transparent optimizations of overheads in optimistic simulations. In *Proceedings of the 1992 Winter Simulation Conference*, pages 637–645, 1992.

[BND89] Amotz Bar-Noy and Danny Dolev. Shared memory vs message passing in an asynchronous distributed environment. In *Proceedings of the 8th ACM Symposium on Principles of Distributed Computing*, pages 307–318. ACM, 1989.

[CM79] K.M. Chandy and J. Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, 5(9):440–452, Sept 1979.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, Massachusetts, 1988.

[DR90] P.M. Dickens and P.F. Reynold. SRADS with local rollback. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 161–164, Jan 1990.

[Fuj90] R. Fujimoto. Parallel distributed discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[Gaf88] A. Gafni. Rollback mechanisms for optimistic distributed simulation systems. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 61–67, July 1988.

[HW91] M. P. Herlihy and W. E. Weihl. Hybrid Concurrency Control for Abstract Data Types. *Journal of Computer and System Sciences*, 43(1):25–61, August 1991. Special issue on the 7th Annual ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems, March 21-23, 1988.

[Jef85] David Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[Kor83] H. F. Korth. Locking primitives in a database system. *Journal of the ACM*, 30(1):55–79, Jan 1983.

[LA93] H.V. Leong and D. Agrawal. Using message semantics to reduce rollback in optimistic message logging recovery schemes. Technical report, University of California at Santa Barbara, Department of Computer Science, Santa Barbara, CA93106, 1993. To appear.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LH89] Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems*, 7(4):321–359, Nov 1989.

[Mis86] J. Misra. Distributed discrete-event simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[SS84] P.M. Schwarz and A.Z. Spector. Synchronizing shared abstract types. *ACM Transactions on Computer Systems*, 2(3):223–250, Aug 1984.

[SSH89] L. Sokol, B. Stucky, and V. Hwang. MTW: A control mechanism for parallel discrete simulation. In *Proceedings of the 1989 International Conference on Parallel Processing*, volume III, pages 250–254. IEEE, 1989.

[Ste91] J.S. Steinman. SPEEDES: Synchronous parallel environment for emulation and discrete event simulation. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 95–103, Jan 1991.

[Wei89] William E. Weihl. The impact of recovery on concurrency control. In *Proceedings of the 8th ACM Symposium on Principles of Database Systems*, pages 259–269. ACM, 1989.

[Wes88] D. West. Optimizing time warp: Lazy rollback and lazy re-evaluation. Master's thesis, University of Calgary, Department of Computer Science, Jan 1988.

Appendix

A Proof of Correctness

We will prove that the protocol using the extended definition of α_1 utilizing both forward commutativity and right-backward commutativity and the prescribed definition of α_2 for processing an out-of-order message m^+ and an anti-message m^- is correct, with respect to the correctness criteria defined in this section. The correctness of the protocol in Figure 5 follows naturally from this result, when the use of forward commutativity is ignored.

A.1 The Correctness Criteria

The system execution X can be characterized by a partial order $<$ on a set of events $\mathbf{E}$. An event $e \in \mathbf{E}$ can be the receipt of a message, a local computation step or the sending of a message. In the context of an object, an event can be either an operation invocation, a local computation step or the sending of a response. An execution X is said to be *correct* if this partial order of events is consistent with the *happen-before* relation [Lam78] on $\mathbf{E}$. When we assume that the RVT of any message is always greater than its SVT, this correctness criteria can be achieved by processing the messages in RVT order [Jef85]. The TWM ensures this correctness by rolling back when a message of smaller RVT is received. Let the global virtual time (GVT) be defined [Jef85]:

> The GVT is the minimum of local virtual times at all entities and the send virtual time of all messages that have been sent but not yet been processed.

The "committed" portion of an execution (i.e. the part of execution with virtual time smaller than GVT) forms a correct execution. Therefore, a correct execution has already masked out every anti-message.

Define the *external behavior* of an entity E (an entity can be either an active process or an object) at GVT t to be the set of messages (with their timestamps) sent by E up to time t and the *internal behavior* of E to be its state. Two entities E_i and E_j are *indistinguishable* at GVT t if and only if their external behaviors and internal behaviors are the same at t. Two system executions X_1 and X_2 are *indistinguishable* at GVT t if and only if all entities are indistinguishable at t.

A system execution X is said to be *correct with respect to behavior* at GVT t if and only if there exists a correct system execution X' such that for all $t' > t$, X and X' are indistinguishable at t'. The notion of correctness with respect to behavior allows some object states to differ, but ensures this difference can never be observed in the future. All correct system executions are correct with respect to behavior. Our protocol generates system executions that are correct with respect to behavior. The TWM generates only correct system executions. Our protocol allows some internal object states at virtual times smaller than the current GVT to differ. This is not allowed in the TWM.

A.2 Processing an Out-of-order Message

As the protocol states, α_1 can be represented as the concatenation of forward commute subsequences f_i and backward commute subsequences b_i. We use the convention f, g, h, b to represent sequences of operations and p, q to represent operations. We also introduce the infix relation $\equiv$ to stand for $equiv()$, whenever it is more convenient. We adopt the convention that whenever two sequences are said to be equivalent and one of them is valid, we implicitly mean the other is also valid. This is straightforward from the definition of equivalence of sequences. Let $\mathcal{I}_{ins}$ and $\mathcal{I}_{del}$ denote the invalidated-by relations and $(q, p) \in \mathcal{I}$ denote q is invalidated-by p.

Lemma 1 *If $valid(h \cdot p)$, $valid(h \cdot f)$ and $\forall q \in f, (p, q) \in \mathcal{FC}$, then $valid(h \cdot f \cdot p)$ and $equiv(h \cdot f \cdot p, h \cdot p \cdot f)$.*

Proof.
By induction on the length of f.

Induction hypothesis:
$valid(h \cdot p) \wedge valid(h \cdot f) \wedge \forall q \in f, (p, q) \in \mathcal{FC} \Rightarrow valid(h \cdot f \cdot p) \wedge equiv(h \cdot f \cdot p, h \cdot p \cdot f)$.

Basis: $|f| = 0$, which is trivially true.

Induction step: Let the hypothesis hold for $f = q_1 \cdot ... \cdot q_k$. Consider $f = q_1 \cdot ... \cdot q_k \cdot q_{k+1}$. Given $valid(h \cdot q_1 \cdot ... \cdot q_k \cdot q_{k+1})$, it follows that $valid(h \cdot q_1 \cdot ... \cdot q_k)$, since valid sequences are prefix-closed. Also, p commutes forward with all q_1 to q_k. By induction hypothesis, $valid(h \cdot q_1 \cdot ... \cdot q_k \cdot p)$ $\diamond$(A.1) and $h \cdot q_1 \cdot ... \cdot q_k \cdot p \equiv h \cdot p \cdot q_1 \cdot ... \cdot q_k$ $\diamond$(A.2). Given $valid(h \cdot q_1 \cdot ... \cdot q_k \cdot q_{k+1})$ in the induction step, by (A.1) and from $(p, q_{k+1}) \in \mathcal{FC}$, we know $valid(h \cdot q_1 \cdot ... \cdot q_k \cdot q_{k+1} \cdot p)$ and $h \cdot q_1 \cdot ... \cdot q_k \cdot q_{k+1} \cdot p \equiv h \cdot q_1 \cdot ... \cdot q_k \cdot p \cdot q_{k+1} \equiv h \cdot p \cdot q_1 \cdot ... \cdot q_k \cdot q_{k+1}$ (by A.2). □

Lemma 2 *If $valid(h \cdot b \cdot p)$ and $\forall q \in b, (p, q) \in \mathcal{RBC}$, then $equiv(h \cdot b \cdot p, h \cdot p \cdot b)$.*

Proof.
Again, by induction on the length of b.

Induction hypothesis: $valid(h \cdot b \cdot p) \wedge \forall q \in b, (p, q) \in \mathcal{RBC} \Rightarrow equiv(h \cdot b \cdot p, h \cdot p \cdot b)$.

Basis: $|b| = 0$, which is trivially true.

Induction step: Suppose the hypothesis holds for $b = q_1 \cdot ... \cdot q_k$. Consider $b = q_1 \cdot ... \cdot q_k \cdot q_{k+1}$. Given that p right commutes backward with q_{k+1} and $valid(h \cdot q_1 \cdot ... \cdot q_k \cdot q_{k+1} \cdot p)$, we have $h \cdot q_1 \cdot ... \cdot q_k \cdot q_{k+1} \cdot p \equiv h \cdot q_1 \cdot ... \cdot q_k \cdot p \cdot q_{k+1}$ $\diamond$(A.3) and $valid(h \cdot q_1 \cdot ... \cdot q_k \cdot p \cdot q_{k+1})$. Therefore $valid(h \cdot q_1 \cdot ... \cdot q_k \cdot p)$ (prefix-closure) $\diamond$(A.4). By induction hypothesis, (A.4) and the fact that p right commutes backward with each of the $q_1, ..., q_k$, we have $h \cdot q_1 \cdot ... \cdot q_k \cdot p \equiv h \cdot p \cdot q_1 \cdot ... \cdot q_k$ $\diamond$(A.5). Thus $h \cdot q_1 \cdot ... \cdot q_k \cdot q_{k+1} \cdot p \equiv h \cdot q_1 \cdot ... \cdot q_k \cdot p \cdot q_{k+1}$ (by A.3) $\equiv h \cdot p \cdot q_1 \cdot ... \cdot q_k \cdot q_{k+1}$ (by A.5). □

Theorem 1 *If* $valid(h \cdot p)$, $valid(h \cdot f_1 \cdot b_1 \cdot p)$, ..., $valid(h \cdot f_1 \cdot b_1 \cdot ... \cdot f_n \cdot b_n \cdot p)$ *and* $\forall q \in f_i, (p,q) \in \mathcal{FC}$ *and* $\forall q \in b_i, (p,q) \in \mathcal{RBC}$*, then* $equiv(h \cdot f_1 \cdot b_1 \cdot ... \cdot f_n \cdot b_n \cdot p, h \cdot p \cdot f_1 \cdot b_1 \cdot ... \cdot f_n \cdot b_n)$.

Proof.

By induction on the number of forward/backward subsequences.

Induction hypothesis:

$valid(h \cdot p) \wedge \forall i \in [1..n](valid(h \cdot f_1 \cdot b_1 \cdot ... \cdot f_i \cdot b_i \cdot p) \wedge \forall q \in f_i, (p,q) \in \mathcal{FC} \wedge \forall q \in b_i, (p,q) \in \mathcal{RBC})$
$\Rightarrow equiv(h \cdot f_1 \cdot b_1 \cdot ... \cdot f_n \cdot b_n \cdot p, h \cdot p \cdot f_1 \cdot b_1 \cdot ... \cdot f_n \cdot b_n)$.

Basis: $n = 0$, it is trivial that $equiv(h \cdot p, h \cdot p)$.

Induction step: Suppose the hypothesis is true for $n = k$. Consider the case for $n = k+1$. By induction hypothesis for $n = k$, $h \cdot f_1 \cdot b_1 \cdot ... \cdot f_k \cdot b_k \cdot p \equiv h \cdot p \cdot f_1 \cdot b_1 \cdot ... \cdot f_k \cdot b_k$ $\diamond$(A.6). Let $h' = h \cdot f_1 \cdot b_1 \cdot ... \cdot f_k \cdot b_k$ and $h'' = h' \cdot f_{k+1}$. We are given that $valid(h' \cdot p)$ and $valid(h' \cdot f_{k+1} \cdot b_{k+1} \cdot p)$. Therefore $valid(h' \cdot f_{k+1})$ (prefix-closed). Given that $\forall q \in f_{k+1}, (p,q) \in \mathcal{FC}$, by lemma 1, we know $valid(h' \cdot f_{k+1} \cdot p)$ and $h' \cdot p \cdot f_{k+1} \equiv h' \cdot f_{k+1} \cdot p$ $\diamond$(A.7). We are given that $valid(h'' \cdot b_{k+1} \cdot p)$ and $\forall q \in b_{k+1}, (p,q) \in \mathcal{RBC}$, it follows from lemma 2 that $h'' \cdot b_{k+1} \cdot p \equiv h'' \cdot p \cdot b_{k+1} \equiv h' \cdot f_{k+1} \cdot p \cdot b_{k+1} \equiv h' \cdot p \cdot f_{k+1} \cdot b_{k+1}$ (by A.7) $\equiv h \cdot p \cdot f_1 \cdot b_1 \cdot ... \cdot b_k \cdot f_{k+1} \cdot b_{k+1}$ (by A.6) $\square$

Our protocol attempts to identify the sequence α_1 such that it is composed of alternating f_i and b_i, with each f_i and b_i satisfying the relevant commutativity properties. The responses at the junctions between b_i and f_{i+1} are also verified. By theorem 1, the two sequences are equivalent. Hence all responses generated during the original execution remain valid and all messages sent during the interval are the same. This implies that the external behavior of the object O is the same as it should be at the LVT of the object after the last operation of α_1 (virtual time β_1). Sequence equivalence also implies that the internal behavior remains unchanged. The object is therefore indistinguishable from what it should be at virtual time β_1, if the message m for p were processed in the RVT order, as in the TWM.

Now consider the second part α_2, defined with the invalidated-by relation.

Theorem 2 *If* $valid(h \cdot p)$ *and* $valid(h \cdot g)$ *and* $\forall q \in g, (q,p) \notin \mathcal{I}_{ins}$*, then* $valid(h \cdot p \cdot g)$.

Proof.

From propositional calculus, we know that $\neg(p \wedge q \wedge \neg r) \Leftrightarrow (p \wedge q \Rightarrow r)$. This allows us to use the definition of the invalidated-by relation in another manner:

q is *not* invalidated-by p if and only if $\forall g, h, valid(h \cdot p \cdot g) \wedge valid(h \cdot g \cdot q) \Rightarrow valid(h \cdot p \cdot g \cdot q)$. Perform induction on the length of g.

Induction hypothesis: $valid(h \cdot p) \wedge valid(h \cdot g) \wedge \forall q \in g, (q,p) \notin \mathcal{I}_{ins} \Rightarrow valid(h \cdot p \cdot g)$.

Basis: $|g| = 0$, trivial.

Induction step: Suppose hypothesis holds for $g = q_1 \cdot q_2 \cdot \ldots \cdot q_k$. Consider $g = q_1 \cdot q_2 \cdot \ldots \cdot q_k \cdot q_{k+1}$. We know $valid(h \cdot p)$ and $valid(h \cdot q_1 \cdot \ldots \cdot q_k \cdot q_{k+1})$ (which implies $valid(h \cdot q_1 \cdot \ldots \cdot q_k)$), and all q_1 to q_k are *not* invalidated by p $\diamond$(A.8). By hypothesis and (A.8), $valid(h \cdot p \cdot q_1 \cdot \ldots \cdot q_k)$. Given that $valid(h \cdot q_1 \cdot \ldots \cdot q_k \cdot q_{k+1})$, the fact q_{k+1} is not invalidated by p leads us to $valid(h \cdot p \cdot q_1 \cdot \ldots \cdot q_k \cdot q_{k+1})$. □

Our protocol calls for a sequence such that the insertion of operation p does not invalidate all the responses generated previously. From theorem 1, we know $equiv(\alpha_0 \cdot p \cdot \alpha_1, \alpha_0 \cdot \alpha_1 \cdot p)$. From theorem 2, we know $valid(\alpha_0 \cdot \alpha_1 \cdot p \cdot \alpha_2)$. This implies $valid(\alpha_0 \cdot p \cdot \alpha_1 \cdot \alpha_2)$. Thus all responses generated in α_2 before were valid, with respect to the insertion of out-of-order operation p. External behavior of object O is same as before. Furthermore, state-reconstruction can start after α_1, owing to the equivalence of states for the sequence α_1. After the re-execution, the internal behavior is guaranteed to be the same as that of a correct execution. Thus O is indistinguishable at the LVT β_2 after the last operation in α_2 from what it should have been, if the out-of-order message were processed in the prescribed order as in the TWM. All other entities are not affected and thus the execution is indistinguishable from the correct one.

A complete rollback is required after α_2 since there is no guarantee of state equivalence or validity of operation sequences. All states and responses are generated afresh according to the correct ordering. The resulting execution is therefore correct, with respect to the TWM. The existence of an operation q such that $equiv(\alpha_0 \cdot \alpha_1 \cdot p \cdot \alpha_2 \cdot \alpha_3 \cdot q, \alpha_0 \cdot \alpha_1 \cdot \alpha_2 \cdot \alpha_3 \cdot q)$ implies $equiv(\alpha_0 \cdot p \cdot \alpha_1 \cdot \alpha_2 \cdot \alpha_3 \cdot q, \alpha_0 \cdot \alpha_1 \cdot \alpha_2 \cdot \alpha_3 \cdot q)$. All future executions are identical from q onwards. The protocol calls for a stop for re-processing, which still guarantees correctness of the execution, since O is now indistinguishable from before. Similarly, all other entities are indistinguishable since they will receive anti-messages and start the protocol themselves. Hence the proposed protocol is correct (with respect to behavior) in the processing of an out-of-order message.

A.3 Processing an Anti-message

An anti-message m^- carries the objective of revoking an operation on an object O. We assume that the operation to be withdrawn has an inverse operation. Let the operation to be withdrawn be p and the inverse operation be p^-. The inverse operation satisfies the identity property

$\forall h, equiv(h, h \cdot p \cdot p^-) \wedge equiv(h, h \cdot p^- \cdot p)$.

The proposed protocol dictates α_1 to be represented as a backward commute portion b followed by a forward commute portion f. The following results show that the protocol is correct for processing an anti-message.

Lemma 3 *If $valid(h \cdot p \cdot f)$ and $\forall q \in f, (q, p^-) \in \mathcal{FC}$, then $valid(h \cdot f)$, $valid(h \cdot p \cdot f \cdot p^-)$ and $equiv(h \cdot f, h \cdot p \cdot f \cdot p^-)$.*

Proof.
Let $h' = h \cdot p$ and let the operation to be considered be p^-. We know that forward commutativity is a symmetric relation. Thus $\forall q \in f, (q, p^-) \in \mathcal{FC}$ is equivalent to $\forall q \in f, (p^-, q) \in \mathcal{FC}$ and we

know that $h' \cdot p^- = h \cdot p \cdot p^-$ is valid. By lemma 1, we have $valid(h' \cdot f \cdot p^-)$ and $h' \cdot f \cdot p^- \equiv h' \cdot p^- \cdot f$. Hence $valid(h \cdot p \cdot f \cdot p^-)$ and $h \cdot p \cdot f \cdot p^- \equiv h \cdot p \cdot p^- \cdot f \equiv h \cdot f$, and $valid(h \cdot f)$. □

Lemma 4 *If $valid(h \cdot p \cdot b)$ and $\forall q \in b, (q, p) \in \mathcal{RBC}$, then $equiv(h \cdot p \cdot b, h \cdot b \cdot p)$.*

Proof.

By induction on the length of b.

Induction hypothesis: $valid(h \cdot p \cdot b) \wedge \forall q \in b, (q, p) \in \mathcal{RBC} \Rightarrow equiv(h \cdot p \cdot b, h \cdot b \cdot p)$.

Basis: $|b| = 0$, which is trivially true.

Induction step: Suppose the hypothesis holds for $b = q_1 \cdot ... \cdot q_k$. Consider $b = q_1 \cdot ... \cdot q_k \cdot q_{k+1}$. Given that $valid(h \cdot p \cdot q_1 \cdot ... \cdot q_k \cdot q_{k+1})$, we have $valid(h \cdot p \cdot q_1 \cdot ... \cdot q_k)$. With the fact that $q_1, ..., q_k$ all right commute backward with p, we know by induction hypothesis that $h \cdot p \cdot q_1 \cdot ... \cdot q_k \equiv h \cdot q_1 \cdot ... \cdot q_k \cdot p$. Since $valid(h \cdot p \cdot q_1 \cdot ... \cdot q_k \cdot q_{k+1})$, it follows that $h \cdot p \cdot q_1 \cdot ... \cdot q_k \cdot q_{k+1} \equiv h \cdot q_1 \cdot ... \cdot q_k \cdot p \cdot q_{k+1}$ and both are valid $\diamond$(A.9). Taking $h' = h \cdot q_1 \cdot ... \cdot q_k$ and the fact that $(q_{k+1}, p) \in \mathcal{RBC}$, we have $h \cdot q_1 \cdot ... \cdot q_k \cdot p \cdot q_{k+1} \equiv h \cdot q_1 \cdot ... \cdot q_k \cdot q_{k+1} \cdot p$. By (A.9), $h \cdot q_1 \cdot ... \cdot q_k \cdot q_{k+1} \cdot p \equiv h \cdot p \cdot q_1 \cdot ... \cdot q_k \cdot q_{k+1}$. □

Theorem 3 *If $valid(h \cdot p \cdot b \cdot f)$ and $\forall q \in f, (q, p^-) \in \mathcal{FC}$ and $\forall q \in b, (q, p) \in \mathcal{RBC}$ then $valid(h \cdot b \cdot f)$, $valid(h \cdot p \cdot b \cdot f \cdot p^-)$ and $equiv(h \cdot p \cdot b \cdot f \cdot p^-, h \cdot b \cdot f)$.*

Proof.
Given that $valid(h \cdot p \cdot b \cdot f)$, we have $valid(h \cdot p \cdot b)$. Knowing that $\forall q \in b, (q, p) \in \mathcal{RBC}$ and by lemma 4, we have $valid(h \cdot b \cdot p)$ and $equiv(h \cdot b \cdot p, h \cdot p \cdot b)$. Thus $equiv(h \cdot b \cdot p \cdot f, h \cdot p \cdot b \cdot f)$ $\diamond$(A.10). Since $valid(h \cdot b \cdot p \cdot f)$ and it is given that $\forall q \in f, (q, p^-) \in \mathcal{FC}$, we have by lemma 3 $valid(h \cdot b \cdot f)$ and $h \cdot b \cdot f \equiv h \cdot b \cdot p \cdot f \cdot p^- \equiv h \cdot p \cdot b \cdot f \cdot p^-$ (by A.10). Furthermore, $h \cdot p \cdot b \cdot f \cdot p^-$ is valid (by definition of sequence equivalence). □

The protocol identifies the sequence α_1 whose segments satisfy the commutativity property. By theorem 3, the sequence is equivalent to the one with the operation p withdrawn at the end and both are valid. This means that all responses were correct and the withdrawal of p may be performed at the end. The anti-message signifies that p should not be performed. Thus the "correct" sequence should be $\alpha_0 \cdot \alpha_1$. The equivalence results of the theorem imply that the external behavior and internal behavior of O remain unchanged, even when we avoid rolling back most operations. The entity is indistinguishable from its correct version at virtual time β_1 (the LVT after the last operation of α_1).

Next consider the sequence α_2 identified by the invalidated-on-delete relation.

Theorem 4 *If $valid(h_1 \cdot h_2)$, $valid(h_1 \cdot p \cdot h_2 \cdot g)$ and $\forall q \in g, (q, p) \notin \mathcal{I}_{del}$, then $valid(h_1 \cdot h_2 \cdot g)$.*

Proof.

An alternative definition of the invalidated-on-delete relation is:

q is *not* invalidated-on-delete by p if and only if $\forall g, h, valid(h \cdot p \cdot g \cdot q) \wedge valid(h \cdot g) \Rightarrow valid(h \cdot g \cdot q)$. Perform induction on the length of g.

Induction hypothesis: $valid(h_1 \cdot h_2) \wedge (h_1 \cdot p \cdot h_2 \cdot g) \wedge \forall q \in g, (q, p) \notin \mathcal{I}_{del} \Rightarrow valid(h_1 \cdot h_2 \cdot g)$.

Basis: $|g| = 0$, trivial.

Induction step: Suppose hypothesis holds for $g = q_1 \cdot q_2 \cdot \ldots \cdot q_k$. Consider $g = q_1 \cdot q_2 \cdot \ldots \cdot q_k \cdot q_{k+1}$. We know $valid(h_1 \cdot p \cdot h_2 \cdot q_1 \cdot \ldots \cdot q_k \cdot q_{k+1})$ (which implies $valid(h_1 \cdot p \cdot h_2 \cdot q_1 \cdot \ldots \cdot q_k)$) $\diamond$(A.11) and given that $valid(h_1 \cdot h_2)$ and all q_1 to q_k are *not* invalidated-on-delete by p $\diamond$(A.12). By hypothesis, (A.11) and (A.12), it is true that $valid(h_1 \cdot h_2 \cdot q_1 \cdot \ldots \cdot q_k)$. Given that $valid(h_1 \cdot p \cdot h_2 \cdot q_1 \cdot \ldots \cdot q_k \cdot q_{k+1})$, the fact q_{k+1} is not invalidated-on-delete by p leads us to $valid(h_1 \cdot h_2 \cdot q_1 \cdot \ldots \cdot q_k \cdot q_{k+1})$. □

We try to find a sequence such that the deletion of p does not invalidate all the responses generated previously. From theorem 3, we know $equiv(\alpha_0 \cdot p \cdot \alpha_1 \cdot p^-, \alpha_0 \cdot \alpha_1)$ and $valid(\alpha_0 \cdot \alpha_1)$. With the fact that $valid(\alpha_0 \cdot p \cdot \alpha_1 \cdot \alpha_2)$ and from theorem 4, we know $valid(\alpha_0 \cdot \alpha_1 \cdot \alpha_2)$. Thus all responses generated in α_2 before were valid, with respect to the revocation of operation p. External behavior is therefore identical to the "correct" version of O. State-reconstruction can start after α_1. This will re-generate the correct internal behavior. As a result, after this step in the protocol, the entity is indistinguishable at virtual time β_2 (the LVT after the last operation in α_2) from what it should be.

A complete rollback is required for α_3 and processing can stop when α_4 is encountered, as in the case of inserting an out-of-order message with the same mechanism. This brings in the correctness ensured by the TWM. Hence, the proposed protocol is correct (with respect to behavior) in the processing of an anti-message.