Formal Specifications and Verification of Message Ordering Properties in a Broadcasting System using Event-B

Divakar Yadav*and Michael Butler[†]

Dependable Systems and Software Engineering Group
School of Electronics and Computer Science
University of Southampton
Southampton SO17 1BJ ,U.K

Abstract

Causal and total order broadcast has been proposed as a mechanism to provide fault tolerance for constructing reliable distributed systems. The use of formal methods to develop a model of a system, specifying critical properties and the verification of them is a way of obtaining better design of dependable services. Event-B is a formal technique which provides a framework for developing mathematical models of distributed systems by rigorous description of the problem, gradually introducing solutions in the refinement steps, and verification of solutions by discharge of proof obligations. In this paper, we present a formal development of a system where processes communicate by broadcast and the messages are delivered following a *causal* and a *total* order. We first present separate models of a broadcast system each for a causal order and a total order. Subsequently, we verify that the models of the system preserves the required ordering properties. Further, we develop a model of a system satisfying both causal and a total order on the messages. Later in the refinement, we outline how these ordering properties can correctly be implemented by the vector clocks. In this approach we discover some interesting invariant properties which describes the relationship of abstract causal and total order with the vector clocks and the sequence numbers.

Keywords : *Distributed System, Formal Method, Verification, Message Ordering, Event-B*

1 Introduction

Distributed systems are difficult to understand, build and reason about due to unavoidable concurrency [20]. In a fully asynchronous message passing system, there is no natural ordering of the messages. In such systems there is no concept of real time and it is assumed that messages are eventually delivered and processes eventually respond, but no assumption on time can be made. Group communication primitives provides higher guarantees on the delivery of messages to different processes. These group communication primitives have been used as a basic building block for the

*Divakar Yadav is a Commonwealth Scholar supported by the Commonwealth Scholarship Commission in the United Kingdom.

[†]Michael Butler's contribution is part of the IST project IST 511599 RODIN (Rigorous Open Development Environment for Complex Systems)

development of reliable fault tolerant distributed applications [12]. The solutions based on group communication are used in the real world. For example, ISIS [8] based solutions are used at the New York Stock Exchange for providing reliable multicast communication, Swiss Electronic Bourse and for developing new generation of the French Air Traffic Control System [13]. These primitives have also been proposed for processing transactions and managing replicated databases [16, 17, 18]. The *total order* [12] broadcast is one primitive which ensures that a message is delivered to the different recipient processes in the same order. The total order alone does not guarantee that messages are delivered in the order they were sent. The *causal order* [9, 12] primitive provide guarantees that delivery order is also consistent with the order they were sent. A causal order on the messages are build when they are sent by a single process (FIFO order) or the different processes(*local order*). In this paper we present a formal development of a system that guarantees a total order conforming with the causal dependencies. This *totally ordered causal broadcast* first builds a causal order then a total order on the messages. Our approach of gradual development of the system is based on the notion of abstraction and refinement. The important feature of this approach is to formally define an abstract global model of a system independent of the architecture and successively refine it to a detailed distributed design in a series of intermediate steps. The work present in the paper constitutes a part of our work on formal development of transactions for replicated databases [28].

Distributed algorithms can be deceptive. An algorithm that looks simple may have complex execution paths and allow unanticipated behavior. Rigorous reasoning about the algorithms is required to ensure that an algorithm achieves what is it supposed to do [20]. The group communication services have been studied as a basic building block for many fault tolerant distributed services, however the application of formal methods for providing clear specifications and proofs of correctness is rare [12]. Some of the important work on application of formal methods to group communication services in order to verify the properties of algorithm are [13, 27]. The work reported in [13] uses I/O automata for the specifications and proves properties about all trace behavior of the automation. A series of invariants relating state variables and reachable states are proved by hand using method of induction. In [27] formal results are provided that defines whether or not a totally ordered protocol provides a causal order. They provide a proof of correctness by doing proofs by hand. Instead, our approach is based on defining properties in abstract model and proving that our model of algorithm is a correct refinement of the abstract model. Also, we use a tool for generating proof obligations and discharging them.

The approach of specifying the system and verification is based on the technique of abstraction and refinement. This technique is supported by the Event-B [2, 22], an event driven approach used together with B Method [1]. This formal technique consists of the following steps :

- Rigorous description of abstract problem.
- Introduce solutions or details in refinement steps to obtain more concrete specifications.
- Verifying that proposed refinements are valid.

This formal approach supports a step-wise development from initial abstract specifications to a detailed design of a system in the refinement steps. Through refinement we verify that the design of detailed system conforms to the abstract specifications. We have used the Click'n'Prove [4] B tool for proof obligation generation and for discharging proof obligations. The tool generate proof obligations for refinement and consistency checking. These proofs help to understand the complexity of problem and the correctness of the solutions. They also guide us to discover new system invariants which provide a clear insight to the system.

The approach for building causal and total order on the messages is based on the work reported in ISIS [9]. We present separate models of message passing systems each for causal order and a total order. These models of message passing system delivers the messages following a causal order and a total order respectively. We also verify that these models of the system preserves the required ordering properties. Further we develop a model of a system satisfying both causal and a total order on the messages. In the refinement, we show that this model can correctly be refined by a system of vector clocks. In the refinement, the causal order is implemented by using a vector clock while the total order is implemented using a fixed sequencer algorithm [12]. In the further refinement, we verify that a total order can be build by using the services of vector clocks alone thus replacing the need for explicit sequence numbers to be generated by a sequencer. We also discover some interesting properties as invariants which describes the relationship between abstract causal and total order with the vector clocks and the sequence numbers.

The remainder of the paper is organized as follows: Section 2 presents our modelling approach and introduction to Event-B notations , Section 3 informally presents various ordering properties, Section 4 present abstract B specification of causal order properties, Section 5 present abstract B specification of total order properties, Section 6 present the specification of a system which respect both total and a causal order, lastly section 7 concludes the paper.

2 Modelling Approach in Event-B

The Event-B [2, 22] is a formal technique consist of describing rigorously the problem, introduce solutions or details in the refinement steps to obtain more concrete specifications, and verifying that proposed solutions are correct. The system is modelled in terms of an abstract state space using variables with set theoretic types and the events that modify state variables. Event-B, a variant of B [1], was designed for developing distributed systems. In Event-B, the events consists of guarded actions occurring spontaneously rather then being invoked. The invariants state properties that must be *satisfied* by the variables and *maintained* by the activation of the events.

In the refinement steps, guards may be strengthened and the new events may be introduced. Abstract and concrete variables are related through *gluing invariants*. At each refinement step a more concrete specification of the system are obtained. This technique requires the discharge of the proof obligations for *consistency checking* and *refinement checking*. The consistency checking involves showing that a machine preserves the invariants when events are invoked. The refinement checking involves showing that the specifications at each refinement step are valid. We have used the Click'n'Prove [4] B tool for proof management which provides an environment for generation of proof obligations for *consistency checking* and *refinement checking*. This tool also provides an automatic and an interactive prover. The majority of the proof obligations are proved by the automatic prover however some of the complex proof obligations need to be proved interactively. Some of the applications of the B method to distributed systems are modelling web based system [25], verification of one copy equivalence criterion in a distributed database system [28], verification of IEEE 1394 tree protocol distributed algorithm [5], and a general modelling approach for distributed system may be found in [10, 11].

2.1 Event-B Notations

The Event-B notations are based on set theoretic notations and frequently used notations in our models are explained here.

Let A and B be two sets, then the relational constructor ($\leftrightarrow$) defines the set of relations between A and B as :

$$A \leftrightarrow B = \mathbb{P}(A \times B)$$

where $\times$ is cartesian product of A and B. A mapping of element $a \in A$ and $b \in B$ in a relation $R \in A \leftrightarrow B$ is written as $a \mapsto b$.

The *domain* of a relation $R \in A \leftrightarrow B$ is the set of elements of A that R relates to some elements in B defined as :

$$dom(R) = \{a \mid a \in A \wedge \exists b.(b \in B \wedge a \mapsto b \in R)\}$$

Similarly, the *range* of relation $R \in A \leftrightarrow B$ is defined as set of elements in B related to some element in A defined as :

$$ran(R) = \{b \mid b \in B \wedge \exists a.(a \in A \wedge a \mapsto b \in R)\}$$

A relation R $\in$ A $\leftrightarrow$ B can be projected on a domain U $\subseteq$ A called *domain restriction*($\lhd$) defined as

$$U \lhd R = \{a \mapsto b \mid a \mapsto b \in R \wedge a \in U\}$$

The *domain anti-restriction* (U $⩤$ R) is defined as :

$$U ⩤ R = \{a \mapsto b \mid a \mapsto b \in R \wedge a \notin U\}$$

The *Relational image* R[U] where U$\subseteq$A is defined as :

$$R[U] = \{b \mid \exists a \cdot a \mapsto b \in R \wedge a \in U\}$$

The *relational inverse* (R^{-1}) of a relation R is defined as :

$$R^{-1} = \{b \mapsto a \mid a \mapsto b \in R\}$$

If $R_0 \in A \leftrightarrow B$ and $R_1 \in A \leftrightarrow B$ are relations defined on set A and B, the *relational over-ride* operator ($R_0 \mathbin{\lhd\!\!\!-} R_1$) replaces mappings in relation R_0 by those in relation R_1.

$$R_0 \mathbin{\lhd\!\!\!-} R_1 = (dom(R_1) ⩤ R_0) \cup R_1$$

A *function* is a relation with certain restrictions. The function may be a partial function ($\nrightarrow$) or a total function($\rightarrow$).

A *partial function* from set A to B ($A \nrightarrow B$) is a relation which relates an element in A to *at most* one element in B.

A *total function* from set A to B ($A \rightarrow B$) is a partial function where dom(f)=A i.e. each element of set A is related to exactly one element of set B. Given $f \in A \nrightarrow B$ and $a \in dom$(f), *f(a)* represents the unique value that a is mapped to by *f*.

2.2 Event-B System

The mathematical foundations for development of event based system in B is discussed in [3]. An abstract machine consists of sets, constants and variables clause modelled as set theoretic constructs. The invariants and properties are defined as first order predicates. The event system is defined by its *state* and contain number of *events*. The state is defined by variables. The constants and variables are constrained by the conditions defined in the properties and invariant clause known as invariant properties of the system. Each event in the abstract model is composed of a guard and an action. The events are modelled using generalized substitution which include construct like assignment (*x:= E(x)*) and guarded statement (*WHEN G THEN S END*). A typical abstract machine may be outlined as below.

```
MACHINE            M
SETS               S_1,S_2,S_3...
CONSTANTS          C
PROPERTIES         P
VARIABLES          v_1,v_2,v_3...
INVARIANTS          I
INITIALISATION      init
EVENTS
      E1 ≅ WHEN G_1  THEN S_1  END;
      E2 ≅ WHEN G_1  THEN S_1  END;
       .......
END
```

In the guarded statement(*WHEN G THEN S END*), the guard(G) of the events are expressed as first order predicate. The actions of an events are specified as simultaneous assignment of state variable using substitution statements(S). The events occur spontaneously whenever their guard holds (true) and they are executed atomically. After building a model of a system as abstract machine, it must be proved that a system is consistent with respect to the invariant properties of the system. The consistency of the machine is shown by proving that each event of the system preserves the invariant.

In an incremental development approach for system modelling, we begin with abstract definition of problem. The system in build in several stages by gradually introducing the details in the refinement steps. At every refinement step we verify that proposed refinements are valid. An abstract machine can be refined by adding new events and new variable. A refined system state must relate to the abstract one by *abstraction relation*. This abstraction relation is defined by a invariant known as *Gluing Invariant*. This invariant defines relationship between abstract state variables and concrete(refined) state variables. More precisely, if a statement S that acts on variable x, is refined by another statement T that acts on variable y under invariants I then we write $S \sqsubseteq_I T$. The invariant I is called the *gluing invariant* and it defines the relationships between x and y. The each event of abstract model is refined to one or more corresponding concrete event. A concrete event is said to refine corresponding abstract one, if it is obtained by strengthening the guards of abstract one and the gluing invariant is preserved by joined action of both event.

Replacing the abstract variable by the concrete variable in the refinement results in proof obligations generated by the B tool. These proof obligations are associated with the events in the refinement. The B Tool help to factorize the large and complex proof obligations in to simpler proof obligations. In most cases majority of the proof are proved by the automatic prover, however in some cases we need to prove them by interaction. The B Tool also remembers the proved and unproved proofs in the form of proof tree. In some cases, in order to prove unproved proof obligations we may have to add gluing invariants to the model. In these cases the unproved proof obligations guide us to construct the gluing invariants. The addition of new gluing invariants further generate the proof obligations which may require addition of new gluing invariants. After several stages of invariant strengthening we arrive at a set of invariants which are sufficient to discharge all proof obligations. Addition of an appropriate invariant is a key to proving the correctness of the refinement. In this approach not only proof obligations and interactive prover together guide constructing new gluing invariants, but also has a consequence that the form of gluing invariant closely matches the form of proof obligations thereby making the mechanical proof much easier and in many cases completely automatic.

3 Ordering Properties

In context of ordered broadcast, Hadzilacos and Toueg [14] defines that a Reliable Broadcast satisfies following properties.

- *Validity*: If a correct process broadcasts a message m, then it eventually delivers m.
- *Agreement*: All correct processes delivers a same set of message, i.e. if a process delivers a message m then all correct processes eventually delivers m.
- *Integrity* : No spurious messages are ever delivered, i.e., for any message m, every correct process delivers m at most once and only if m was previously broadcast by $sender(m)$.

A Reliable Broadcast is defined in terms of two primitives called *broadcast* and *deliver*. A reliable broadcast imposes no restriction on the order in which messages are delivered to the processes. However, many application requires a stronger notion of reliable broadcast that provide higher order guarantees on message delivery. A reliable broadcast can be used to deliver messages to the processes following a *FIFO Order*, *Local Order*, *Causal Order* or a *Total Order* providing higher ordering guarantees on the message delivery. Various definitions of ordering properties have been discussed in [6, 9, 12, 26]. An informal specifications of these ordering properties are given below.

FIFO Order

If a particular process broadcasts a message M1 before it broadcasts a message M2, then each recipient process delivers M1 before M2.

A *FIFO Broadcast* is defined as a reliable broadcast that delivers the messages in *FIFO order*. As shown in Fig. 1, process *P1* first broadcasts *M1* followed by *M2*. Each recipient process delivers message *M1* before *M2* respecting FIFO order. The FIFO order is said to be preserved by the system if all processes deliver *M1* before delivering *M2*. The delayed message *M1* shown as dotted line violates the FIFO order.

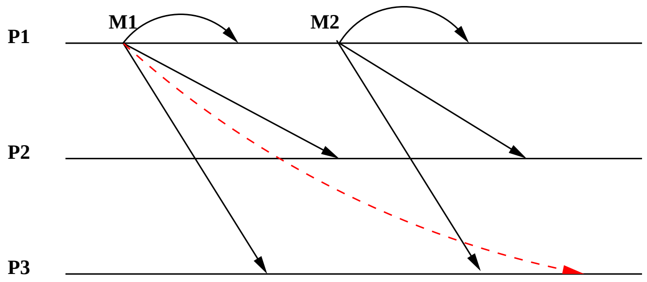


Figure 1: FIFO order

Local Order

If a process delivers M1 before broadcasting the message M2, then each recipient process delivers M1 before M2.

As shown in Fig. 2, process *P2* delivers *M1* before it broadcasts *M2*. Each recipient process delivers message *M1* before *M2* respecting local order. The local order is said to be preserved by the system if all processes deliver *M1* before delivering *M2*. The delayed message *M1* shown as dotted line violates the local order.

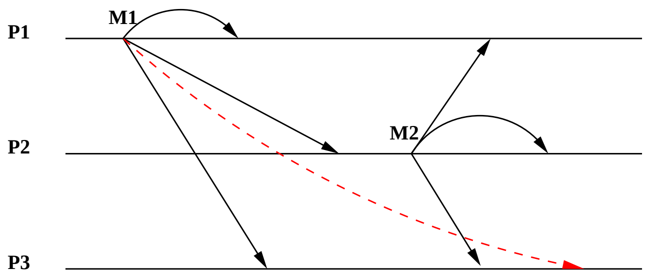


Figure 2: Local order

Causal Order

If the broadcast of a message M1 causally precedes the broadcast of a message M2, then no process delivers M2 unless it has previously delivered M1.

The notion of causality is based on *causal precedence relation* ($\rightarrow$) defined by the Lamport [19] as follows. Let e and f be two events in the distributed system, the causal precedence relation ($e \rightarrow f$) holds, if and only if,

- e and f are the events occurring in the same process and e *happened before* f.
- e and f are the event of *message send* and *message receive* respectively occurring in the different processes.
- There exist another event g, such that $e \rightarrow g$ and $g \rightarrow f$. (*Transitivity*)

The *causal precedence* relation defines an irreflexive partial ordering on the set of events. This relationship can be extended to define causality among the messages. A message m_i *causally precedes* m_j if either of following holds,

- the broadcast event of m_i *causally precedes* broadcast of m_j.
- the receive event of m_i *causally precedes* broadcast of m_j.

The causal order is defined by combining the properties of both FIFO and Local order [14]. A *Causal Order Broadcast* is a reliable broadcast that satisfies *causal order* requirement. A causal order broadcast delivers the messages respecting their causal precedence, however if the broadcast of any two message is not related by causal precedence then it does not impose any requirement on the order they can be delivered.

Total Order

If two process P1 and P2 both delivers the messages M1 and M2 then P1 deliver M1 before M2 if only if P2 also delivers M1 before M2.

A *Total Order Broadcast*[1] is a reliable broadcast that satisfies *total order* requirement. The *Agreement* and *Total Order* requirements of Total Order Broadcast imply that all correct processes eventually deliver the same *sequence* of messages [14].

Since a total order define a arbitrary ordering on the delivery of messages, it does not satisfy causal relations. Consider the following two cases given in the Fig. 3 and Fig. 4. In the first case, the messages are delivered conforming to both causal and total order, however, in the second the delivery order respect total order but violates causal order.

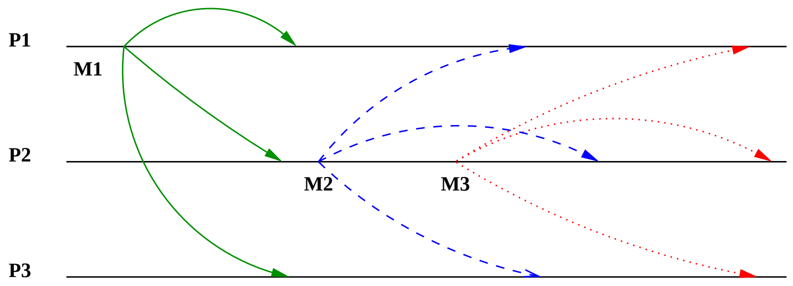


Figure 3: Total Order and a Causal Order

As shown in Fig. 3, broadcast of message *M1 causally precedes* broadcast of *M2*, each recipient process delivers *M1* before delivering *M2*. Similarly, the broadcast of message *M2 causally precedes* broadcast of *M3*, each recipient process delivers *M2* before delivering *M3*. Therefore, the system delivers messages respecting the *causal order*. It can also be noticed that all processes delivers the message in the same sequence i,e.(*m1*,*m2*, followed by *m3*), the delivery order also conforms to *total order*. A reliable broadcast which satisfies both causal order and total order is called *causally and totally order broadcast*[2].

As shown in Fig. 4, all processes deliver the same sequence of messages, i.e., each process delivers *M1* followed by *M3*, and lastly *M2*. Thus the delivery order conforms to the *total order* property, however the delivery order does not respect the causal order for the following reason. Since the broadcast of *M2 causally precedes* the broadcast of *M3*, each recipient must deliver *M2* before delivering *M3*. It can be noticed that each process deliver *M3* before delivering *M2* violating the causal order.

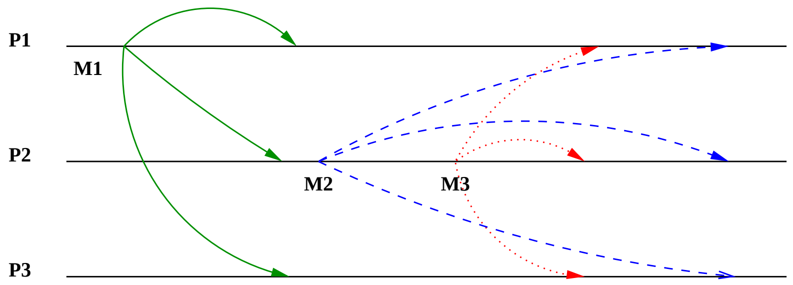


Figure 4: Total Order but not a Causal Order

[1]The *Total Order Broadcast* is also known as Atomic Broadcast. Both of the terms are used interchangeably. However we prefer former as the term *atomic* suggest the *agreement* property rather than *total order*.

[2]A reliable broadcast that satisfies both causal and total order is also called *Causal Atomic Broadcast*.

4 Causal Order Broadcast

In this section we present an incremental development of a system of causal order broadcast consisting of five levels. A brief outline of each level is given below.

- L1 This consist of abstract model of reliable broadcast. In this model a process communicate by broadcast and messages are delivered to the processes only once.
- L2 This is a refinement of abstract model which introduces causal ordering on the messages. In this refinement we outline how an abstract causal order is constructed by the sender.
- L3 In this refinement we introduce the notion of vector clocks. The abstract causal order is replaced by the vector clocks rules. We also discover gluing invariants which defines the relationship of abstract causal order and vector rules.
- L4 In this refinement steps we present the simplification of vector rules updating the vector clock of recipient process. Here we show that instead of updating whole vector of recipient process, a single value is updated.
- L5 This is a simple refinement further simplifying of causal deliver event.

4.1 Abstract Model of Reliable Broadcast

The abstract model of a reliable broadcast system is presented as a B machine in the Fig. 5. The PROCESS and MESSAGE are defined as sets. The brief description of this machine is given as follows.

```
MACHINE          Broadcast
SETS             PROCESS; MESSAGE
VARIABLES        sender , cdeliver
INVARIANT
/* I-1*/            $sender \in MESSAGE \rightarrow\!\!\!\!\!\!+ PROCESS$
/* I-2*/          $\wedge$ $cdeliver \in PROCESS \leftrightarrow MESSAGE$
/* I-3*/          $\wedge$ $ran(cdeliver) \subseteq dom(sender)$

INITIALISATION   $sender := \emptyset \parallel cdeliver := \emptyset$

OPERATIONS
 Broadcast ( $pp \in PROCESS$ , $mm \in MESSAGE$)  $\triangleq$
                 WHEN   $mm \notin dom(sender)$
                 THEN   $sender := sender \cup \{mm \mapsto pp\}$
                 END;

CausalDeliver ($pp \in PROCESS$ , $mm \in MESSAGE$) $\triangleq$
                 WHEN   $mm \in dom(sender)$
                          $\wedge (pp \mapsto mm) \notin cdeliver$
                 THEN   $cdeliver := cdeliver \cup \{pp \mapsto mm\}$
                 END ;
END
```

Figure 5: Abstract Model of Broadcast

The *sender* is a partial function from *MESSAGE* to *PROCESS* defined in invariant *I-1*. The mapping $(m \mapsto p) \in sender$ indicate that message m was sent by process p. The *cdeliver* is a relation between *PROCESS* and *MESSAGE* defined in invariant *I-2*. A mapping of form $(p \mapsto m) \in cdeliver$ indicate that a process p has delivered a message m. The *sender* and *cdeliver* are initialized as empty set.

In our model of reliable broadcast, a *sent* message is also delivered to its sender. It may be noticed that all delivered messages must be messages whose *Message Sent* event is also recorded. This property is defined as invariant *I-3*. The events of sending and casually ordered delivery of messages are modelled as *Broadcast(pp,mm)* and *CausalDeliver(pp,mm)*. When a *Broadcast* event is invoked, the entry of a process and the corresponding message is made to the *sender*. The *CausalDeliver* event is guarded by predicates. These predicates ensures that a process delivers a message whose *Message Sent* event is recorded and the message has not been delivered before. A message is delivered to a process if both conditions are satisfied.

4.2 First Refinement : Introducing Causal Order

In this refinement we introduce the causal ordering on the messages. We also outline how a causal order for the message is constructed by the sender. The refinement of abstract model of broadcast is given in Fig. 6 and Fig. 7. A brief description of the refinement steps are given below.

The abstract causal order is represented by a variable *corder*. A mapping of the form $(m1 \mapsto m2\ ) \in corder$ indicate that message *m1 causally precedes m2* (*Inv I-4*). In order to represent the delivery order of messages at a process, variable *delorder* is used. A mapping $(m1 \mapsto m2) \in delorder(p)$ indicate that process p has delivered *m1* before *m2*(*Inv I-5*). A causal order on the messages can be defined only on those messages whose message sent event is recorded (*Inv I-6*).

REFINEMENT		*CausalOrder*
REFINES		*Broadcast*
VARIABLES		*sender, cdeliver, corder, delorder*
INVARIANT		
/ I-4*/*		$corder \in MESSAGE \leftrightarrow MESSAGE$
/ I-5*/*	$\wedge$	$delorder \in PROCESS \rightarrow\!\!\!\!\!\rightarrow (MESSAGE \leftrightarrow MESSAGE)$
/ I-6*/*	$\wedge$	$dom(corder) \subseteq dom(sender)$
/ I-7*/*	$\wedge$	$ran(corder) \subseteq dom(sender)$
INITIALISATION		$sender := \varnothing \;\; \| \;\; cdeliver := \varnothing$
	$\|$	$corder := \varnothing \;\; \| \;\; delorder := \varnothing$

Figure 6: Causal Order Broadcast : Initialization

The events *Broadcast(pp,mm)* and *CausalDeliver(pp,mm)* respectively models the events of broadcasting a message and the causally ordered delivery of a message. As shown in the operation of the *Broadcast* event, a causal order is built by the sender process following a FIFO order and a Local order. When a process *pp* broadcasts a message *mm*, the variable *corder* is updated by the mappings in $(sender^{-1}[\{pp\}] \times \{mm\})$. This indicate that all message sent by *pp* before broadcasting *mm* causally precedes *mm* conforming to the FIFO order. Similarly, all mappings in $(cdeliver[\{pp\}] \times \{mm\})$ indicate that all messages causally delivered to the process *pp* before broadcasting *mm* also causally precedes *mm* conforming to a local order.

On the occurrence of the *Broadcast* event, variable *sender* is updated with corresponding entries of sender process and the message. The guard $mm \notin \mathrm{dom}(sender)$ ensures that each time a fresh message is broadcasted. In the *CausalDeliver* event, a process *pp* delivers a message *mm* only when all messages which causally precedes *mm* are delivered. The guards of this event also ensures that a message is delivered only once.

```
Broadcast ($pp \in PROCESS$ , $mm \in MESSAGE$ ) $\widehat{=}$
        WHEN   $mm \notin dom(sender)$
        THEN   $corder := corder \cup ( (sender^{-1}[\{pp\}] \times \{mm\})$
                        $\cup ( cdeliver [\{pp\}] \times \{mm\}))$
            || $sender := sender \cup \{mm \mapsto pp\}$
        END;


CausalDeliver ($pp \in PROCESS$ , $mm \in MESSAGE$) $\widehat{=}$
        WHEN   $mm \in dom(sender)$
            $\wedge$ $(pp \mapsto mm) \notin cdeliver$
            $\wedge$ $\forall m.( m \in MESSAGE \wedge (m \mapsto mm) \in corder$
                        $\Rightarrow (pp \mapsto m) \in cdeliver)$
        THEN   $cdeliver := cdeliver \cup \{pp \mapsto mm\}$
            || $delorder(pp) := delorder(pp) \cup ( cdeliver [\{pp\}] \times \{mm\})$
        END
```

Figure 7: Causal Order Broadcast : Events

4.3 Invariant Properties of the Model of Causal Order

After building the model of the abstract causal order our goal was to formally verify that this model preserves the *causal order* properties *informally* defined in the section 3. It state that the delivery order of the messages at a given process must conforms to the abstract causal order among them. Consider following two cases generated by the *Pro B* [21] ,an animator and a model checker for B.

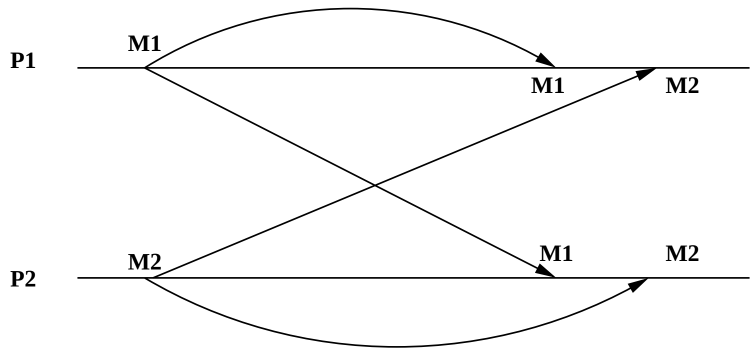


Figure 8: Causal Order : CASE-I

As shown in Fig. 8, messages *M1* and *M2* have same delivery order at processes *P1* and *P2* but have different delivery order as shown in Fig. 9. This is possible

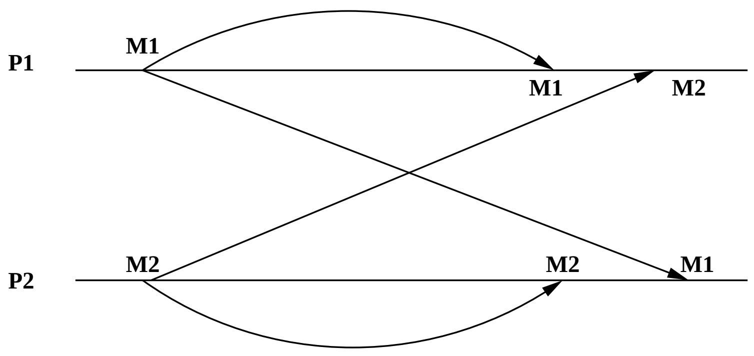


Figure 9: Causal Order : CASE-II

when *M1* and *M2* does not have any causal ordering among them. Also [14] reports that if the broadcast of two messages are not related by causal precedence, a causal broadcast does not impose any requirement on the order they are delivered and the delivery order of any two messages may be different at various processes. Therefore we add following invariant to our model as a *primary invariant.*

$$\begin{array}{l} m1 \mapsto m2 \in corder \\ p \mapsto m2 \in cdeliver \\ \Rightarrow \\ m1 \mapsto m2 \in delorder(p) \end{array}$$

In order to verify that our model also preserves the transitivity property on the messages, we also add following invariant to our model as a primary invariant.

$$\begin{array}{l} m1 \mapsto m2 \in corder \\ m2 \mapsto m3 \in corder \\ \Rightarrow \\ m1 \mapsto m3 \in corder \end{array}$$

The important invariant properties of the model of *causal order broadcast* system are given in Fig. 10 as first order predicates. We have omitted the quantifications over all identifiers to avoid clutter.

4.4 Proof Obligations and Invariant Discovery

In this section we outline how we verify that the model *CausalOrder* given in Fig. 6 and Fig. 7 preserves the *causal ordering* on the messages. We also outline how the proof obligations generated by B Tool and the interactive prover guide us constructing new invariants.

We first add the invariant *Inv-1* given in Fig. 10 to our model. After addition of this invariant to the model, the B Tool generated two proof obligations associated with events *Broadcast* and *CausalDeliver.* These proof obligations were discharged using interactive prover without having to add new invariants.

In the next step, we add invariant *Inv 2* to our model. This invariant state that our model of *Causal Broadcast* preserves *causal precedence* relationship on the messages. When this invariant is added to the model, the Click'n'Prove B Tool generates following complex proof obligation associated with the *Broadcast* event.

	Invariants	**Required By**
*/*Inv-1*/*	$(m1 \mapsto m2) \in corder \wedge (p \mapsto m2) \in cdeliver \Rightarrow (m1 \mapsto m2) \in delorder(p)$	*Broadcast, CausalDeliver*
*/*Inv-2*/*	$(m1 \mapsto m2) \in corder \wedge (m2 \mapsto m3) \in corder \Rightarrow (m1 \mapsto m3) \in corder$	*Broadcast*
*/*Inv-3*/*	$(m1 \mapsto m2) \in corder \wedge (m2 \mapsto p) \in sender \Rightarrow ((m1 \mapsto p \in sender) \cup (p \mapsto m1 \in cdeliver))$	*Broadcast, CausalDeliver*
*/*Inv-4 */*	$(m1 \mapsto m2) \in corder \wedge (p \mapsto m2) \in cdeliver \Rightarrow (p \mapsto m1) \in cdeliver$	*Broadcast, CausalDeliver*

Figure 10: Invariants-I

$$Broadcast(pp, mm)PO1$$
$$\left[\begin{array}{l} Inv2 \wedge \\ mm \notin dom(sender) \wedge \\ m1 \mapsto m2 \in (corder \cup (sender^{-1}[\{pp\}] \times \{mm\}) \cup (cdeliver[\{pp\}] \times \{mm\})) \wedge \\ m2 \mapsto m3 \in (corder \cup (sender^{-1}[\{pp\}] \times \{mm\}) \cup (cdeliver[\{pp\}] \times \{mm\})) \\ \Rightarrow \\ m1 \mapsto m3 \in (corder \cup (sender^{-1}[\{pp\}] \times \{mm\}) \cup (cdeliver[\{pp\}] \times \{mm\})) \end{array}\right]$$

This proof obligation is reduced to following two simple proof obligations using interactive prover.

$$Broadcast(pp, mm)PO2$$
$$\left[\begin{array}{l} m1 \mapsto m2 \in corder \\ m2 \mapsto pp \in sender \\ m1 \mapsto pp \notin sender \\ \Rightarrow \\ m1 \in cdeliver[\{pp\}] \end{array}\right]$$

and

$$Broadcast(pp, mm)PO3$$
$$\left[\begin{array}{l} m1 \mapsto m2 \in corder \\ m2 \mapsto pp \in sender \\ m1 \notin cdeliver[\{pp\}] \\ \Rightarrow \\ m1 \mapsto pp \in sender \end{array}\right]$$

The proof obligation *PO2* generated by the *Broadcast* event state that if a message m1 *causally precedes* m2 i.e., $(m1 \mapsto m2) \in corder$, and that *pp* is sender of *m2* and *m1* was not sent by process *pp* then process *pp* must have delivered *m1*. This corresponds to the property of *local order*. Similarly, the proof obligation *PO3* state that if m1 *causally precedes* m2; and *pp* is sender of *m2* and *pp* have not delivered *m1* then *pp* is sender of *m1*. It can be noticed that this property corresponds to the *FIFO* order. Therefore, to discharge these proof obligations, we add following invariant to the model.

$$\begin{array}{l} m1 \mapsto m2 \in corder \wedge \\ m2 \mapsto p \in sender \\ \Rightarrow \\ ((m1 \mapsto p) \in sender) \cup ((p \mapsto m1) \in cdeliver) \end{array}$$

This invariant is given as *Inv 3* in the Fig. 10. After adding invariant *Inv 3* to the model we discharge the proof obligations *PO2* and *PO3* associated with the *Broadcast* event. However, due to addition of *Inv 3* additional proof obligations associated with *Broadcast* and *CausalDeliver* events are generated. The proof obligation associated with the *Broadcast* event is discharged using interactive prover. The following proof obligation associated with *CausalDeliver* event can not be discharged interactively.

$$\begin{array}{l} CausalDeliver(pp, mm)PO4 \\ \left[\begin{array}{l} Inv\ 3 \wedge \\ m1 \mapsto m2 \in corder \\ pp \mapsto m2 \in cdeliver \\ \Rightarrow \\ m1 \in (sender^{-1}[\{pp\}]) \cup (cdeliver[\{pp\}]) \end{array} \right] \end{array}$$

The *PO4* state that for message *m1* and *m2* where *m1* causally precedes *m2* and a process *pp* has delivered *m2* then *pp* has either delivered *m1* or broadcasted *m1*. On simplifying the PO4 with the hypothesis given as *Inv 2*, it require us to prove following.

$$\begin{array}{l} m1 \mapsto m2 \in corder \wedge \\ p \mapsto m2 \in cdeliver \\ \Rightarrow \\ p \mapsto m1 \in cdeliver \end{array}$$

We add above as an invariant to our model given as *Inv 4* in the Fig. 10. It state that if *m1* causally precedes *m2* then for any process *p* who has delivered *m2* implies that it has also delivered *m1*. After adding invariant *Inv 4* to the model we are able to discharge *PO4*. Addition of *Inv 4* generate new proof obligations associated with *Broadcast* and *CausalDeliver* events. These proof obligations are also discharged interactively using interactive prover. It can be noticed that invariant *Inv 4* also state the causal order correctness criterion and is discovered during invariant strengthening.

We observe that after three iteration of invariant strengthening we arrive at a set of invariants that is sufficient to discharge all proof obligations. By discharging all proof obligations we ensure that this model preserves the *causal precedence* relationship on the messages.

4.5 Second Refinement : Introducing Vector Clocks

In this section we present how abstract causal order can be implemented by vector clocks. The goals of this refinement are given below.

– To replace abstract global variable *corder* with equivalent vector clock rules.

– To refine operation *Broadcast* to generate the vector timestamp of message which is equivalent to global causal order.

– To refine operation *CausalDeliver* to include a mechanism by which early message or messages violating the global causal order may be detected at the recipient process.

In a system of vector clocks [9, 7, 23, 24], every process maintains a vector of size N where N is the total number of processes in the system. Process P_i maintains a vector clock VT_{Pi} where VT_{Pi}(i) is the local logical time at P_i while VT_{Pi}(j) represents the process P_i's latest knowledge of the time at process P_j. More precisely VT_{Pi}(j) (i$\neq$j) represents the time of occurrence of an event at process P_j when the most recent message was sent from P_j to P_i directly or indirectly.

In our model, we use following vector rules [9] to update the vector clock of a process sending or receiving a message and to timestamp a message.

I. While sending a message M from process P_i to P_j, sender process P_i updates its own time(i^{th} entry of vector) by updating VT_{Pi}(i) as VT_{Pi}(i) := VT_{Pi}(i) + 1. The message time stamp VT_M of message M is generated as VT_M(k) := VT_{Pi}(k), $\forall$ k $\in$ (1..N), where N is number of processes in system. Since a process P_i increments its own value only at the time of sending a message, VT_{Pi}(i) indicates number of messages sent out by process P_i.

II. The recipient process P_j delays the delivery of message M until following conditions are satisfied.

   i VT_{Pj}(i)= VT_M(i) - 1

   ii VT_{Pj}(k)$\geq$ VT_M(k), $\forall$k $\in$ (1..N) $\wedge$ (k $\neq$ i).

   The first condition ensures that process P_j has received all but one message sent by process P_i. The second condition ensures that process P_j has received all messages received by sender P_i before sending the message *M.* These conditions ensures global ordering on messages.

III. The recipient process *Pj* updates its vector clock VT_{Pj} at *message receive* event of message M as VT_{Pj}(k) := max (VT_{Pj}(k)$,VT_M$ (k)). Therefore in vector clock of process P_j, VT_{Pj}(i) indicates the number of messages delivered to process P_j sent by process P_i.

This refinement(second refinement) consists of four state variables *sender*, *cdeliver*, *VTP* and *VTM.* The new state variables *VTP* and *VTM* respectively represents vector time of a process and the vector time stamp of a message. These variables are typed as follows.

$$VTP \in PROCESS \rightarrow (PROCESS \rightarrow NATURAL)$$
$$VTM \in MESSAGE \rightarrow (PROCESS \rightarrow NATURAL)$$

These variables are initialized as follows,

$$VTP := PROCESS \times \{PROCESS \times 0\}$$
$$VTM := MESSAGE \times \{PROCESS \times 0\}$$

As shown above, the variables *VTP* and *VTM* are initialized by assigning a array of vector initialized with zero to each process and messages.

The refined specifications of *Broadcast*, and *CausalDeliver* event are given in Fig 11. As shown in the *BroadCast* specifications operations involving abstract variable *corder* are replaced by the vector rules. It can be noticed that at the time of broadcasting a message *mm*, process *pp* increments its own clock value *VTP(pp)(pp)* by one. The *VTP(pp)(pp)* represents the number of messages sent by process *pp.* The modified vector timestamp of process is assigned to message *mm* giving vector timestamp of message *mm.*

As shown in the event *CausalDeliver*, the messages are delivered at a process only if it has delivered all but one message from the sender of that message. Vector timestamp of recipient process and message are also compared to ensure that all

```
BroadCast (pp ∈ PROCESS , mm ∈ MESSAGE) ≙
  WHEN   mm ∉ dom(sender)
  THEN   LET      nVTP   BE
                  nVTP = VTP(pp) ⩤ { pp ↦ VTP(pp)(pp)+1}
         IN       VTM(mm) := nVTP
                  || VTP(pp) := nVTP
         END
      || sender := sender ∪ {mm ↦ pp}
  END ;

CausalDeliver(pp ∈ PROCESS , mm ∈ MESSAGE) ≙
  WHEN      mm ∈ dom(sender)
         ∧  (pp ↦ mm) ∉ cdeliver
         ∧  ∀p.( p ∈ PROCESS ∧ p ≠ sender(mm) ⇒ VTP(pp)(p) ≥ VTM(mm)(p))
         ∧  VTP(pp)(sender(mm)) = VTM (mm)(sender(mm)) - 1
  THEN      cdeliver := cdeliver ∪ {pp ↦ mm}
         || VTP(pp) := VTP(pp) ⩤
                ({q | q ∈ PROCESS ∧ VTP(pp)(q) < VTM(mm)(q)} ◁ VTM(mm))
  END;
```

Figure 11: Refinement with Vector Clocks

messages delivered by the sender of the message before sending it, are also delivered at the recipient process. These conditions are included as a guard in *CausalDeliver* operation. It can be noticed that the guard involving the variable *corder* in the abstract model are replaced by the guards involving comparison of vector timestamp of message and process in the refinement.[3], [4]

4.6 Gluing invariants relating Causal Order and Vector Rules

The replacement of the operations and guards involving variable *corder* in abstract model with operations and guards involving vector clock rule in refinement generates proof obligations. These proof obligations can be discharged interactively using a B Prover after three round of invariant strengthening. A full set of gluing invariants involving abstract causal order and vector clock rules are given in Fig. 12. A brief description of these properties are given below.

- If the vector time of process P is equal or more than vector time stamp of any sent message M then P must have delivered message M. (*Inv-5*)
- For any two messages *m1* and *m2* where *m1 causally precedes m2*, the vector time stamp of *m1* is always less than vector time stamp of *m2*.(*Inv-6*)
- Since *VTP(p)(p)* represent total number of messages sent by process p and *VTM(m)(p)* represent number of messages received by the sender of m from process p before sending m , the number of messages sent by process p will be greater than or equal to the number of messages received by *sender(m)* from p .(*Inv-7*)
- A message whose time stamp is a vector of zero's implies that it is not causally ordered. (*Inv-8*)

[3](f ⩤ g) represents function *f* overridden by *g*.
[4](s ◁ f) represents function *f* is domain restricted by *s*.

– For any two separate processes *p1* and *p2* , knowledge of occurrence of events of p2 at p1 can not be better than knowledge at p2 itself. This property exists in the distributed system where any common clock or memory does not exist. (*Inv-9*)

4.7 Third and Fourth Refinement : Refinement of Causal Deliver Event

It may be recalled that according to *Rule III* of vector clock, in the event of causally ordered delivery of message M sent by the process Pi, the recipient process Pj updates its vector clock VT_{Pj} as, VT_{Pj}(k) := max (VT_{Pj}(k)$, VT_M$ (k)). It can be be noticed that according to *Rule II*, the recipient process P_j delays the delivery of message M until following conditions are satisfied.

i VT_{Pj}(i)= VT_M(i) - 1

ii VT_{Pj}(k)$\geq$ VT_M(k), $\forall$k $\in$ (1..N) $\wedge$ (k $\neq$ i).

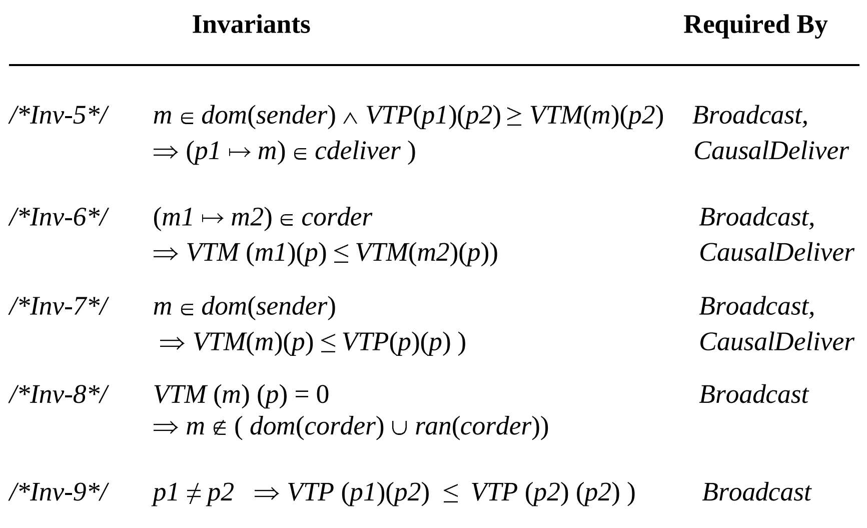
The above conditions are included as guards in the event *CausalDeliver* given in Fig. 11. In the third refinement we show that in the event of delivery of a message, only one value in the vector clock of recipient process which corresponds to the sender process of message, is modified. Consider the following statement of second refinement.

$$VTP(pp) := VTP(pp) \Leftarrow \\ \{(q \mid q \in PROCESS \wedge VTP(pp)(q) < VTM(mm)(q)\} \lhd VTM(mm))$$

The above operation is replaced by the following simplified operation in the third refinement,

$$VTP(pp) := VTP(pp) \Leftarrow \{sender(mm) \mapsto VTM(mm)(sender(mm))\}$$

The above expression state that only one value in the vector clock of the recipient process *pp* corresponding to the sender process of message is updated. The refined *CausalDeliver* event is shown in the Fig. 13.



	Invariants	Required By
/*Inv-5*/	$m \in dom(sender) \wedge VTP(p1)(p2) \geq VTM(m)(p2) \Rightarrow (p1 \mapsto m) \in cdeliver)$	*Broadcast, CausalDeliver*
/*Inv-6*/	$(m1 \mapsto m2) \in corder \Rightarrow VTM(m1)(p) \leq VTM(m2)(p))$	*Broadcast, CausalDeliver*
/*Inv-7*/	$m \in dom(sender) \Rightarrow VTM(m)(p) \leq VTP(p)(p))$	*Broadcast, CausalDeliver*
/*Inv-8*/	$VTM(m)(p) = 0 \Rightarrow m \notin (dom(corder) \cup ran(corder))$	*Broadcast*
/*Inv-9*/	$p1 \neq p2 \Rightarrow VTP(p1)(p2) \leq VTP(p2)(p2))$	*Broadcast*

Figure 12: Invariants-II

```
CausalDeliver(pp ∈ PROCESS , mm ∈ MESSAGE) ≅
  WHEN      mm ∈ dom(sender)
          ∧ (pp ↦ mm) ∉ cdeliver
          ∧ ∀p.( p ∈ PROCESS ∧ p ≠sender(mm) ⇒ VTP(pp)(p) ≥ VTM(mm)(p))
          ∧ VTP(pp)(sender(mm)) = VTM (mm)(sender(mm)) - 1
  THEN      cdeliver := cdeliver ∪ {pp ↦ mm}
          || VTP(pp) := VTP(pp) ⩤ { sender(mm) ↦ VTM(mm)(sender(mm) }
  END;
```

Figure 13: Refined Causal Deliver Event : Third Refinement

This operation is further refined to the following in the fourth refinement which precisely state that only one value in the vector clock of recipient process is updated. The refined operations of model of vector clock implementation of abstract causal order are given in Fig. 14.

$$VTP(pp)(sender(mm)) := VTM(mm)(sender(mm))$$

```
BroadCast (pp ∈ PROCESS , mm ∈ MESSAGE) ≅
  WHEN   mm ∉dom(sender)
  THEN   LET       nVTP   BE
                   nVTP = VTP(pp) ⩤ { pp ↦ VTP(pp)(pp)+1}
         IN        VTM(mm) := nVTP
                 || VTP(pp) := nVTP
         END
       || sender := sender ∪ {mm ↦ pp}
  END ;

CausalDeliver(pp ∈ PROCESS , mm ∈ MESSAGE) ≅
  WHEN      mm ∈ dom(sender)
          ∧ (pp ↦ mm) ∉ cdeliver
          ∧ ∀p.( p ∈ PROCESS ∧ p ≠sender(mm) ⇒ VTP(pp)(p) ≥ VTM(mm)(p))
          ∧ VTP(pp)(sender(mm)) = VTM (mm)(sender(mm)) - 1
  THEN      cdeliver := cdeliver ∪ {pp ↦ mm}
          || VTP(pp) (sender(mm)) := VTM(mm)(sender(mm))
  END;
```

Figure 14: Refined Causal Deliver Event : Fourth Refinement

Proof Statistics

The over all proof statistics is given in Table 1. Approximately sixty eight percent of the proofs were discharged by the automatic prover, the rest were discharged by using interactive prover of B Tool.

5 Total Order Broadcast

In this section we present the incremental development of a system of total order broadcast. It is our assumption that processes communicate by a reliable broadcast.

Machine	Total POs	Completely Automatic	Required Interaction
Abstract Model	14	14	00
Refinement1	43	21	22
Refinement2	47	28	19
Refinement3	06	06	00
Refinement4	02	02	00
Total	112	71	41

Table 1: Proof Statistics- Causal Order Broadcast

The key issues with respect to the total order broadcast algorithms are ; how to build an order? and what information is necessary for defining a total order? In our refinement we show that a total order on the messages can be achieved using a fixed sequencer algorithm [12].

In the sequencer based algorithms, a specific process is elected as a sequencer and become responsible for building a total order. A sequencer process may also takes the role of a *sender* and *destination* in addition to the role of *sequencer*. In a fixed sequencer approach [9, 15], to broadcast a message m, a sender sends m to the sequencer. Upon receiving m, sequencer assigns it a sequence number and send its sequence number to all destinations through control messages. There exist three variants of fixed sequencer algorithms called UB(*Unicast Broadcast*), BB(Broadcast Broadcast) and UUB(*Unicast Unicast Broadcast*). Our mechanism for implementation of total order is based on the *Broadcast Broadcast*(*BB*) variant [9]. The protocol consists of first broadcasting m to all destinations including sequencer, followed by a another broadcast of its sequence numbers by the sequencer.

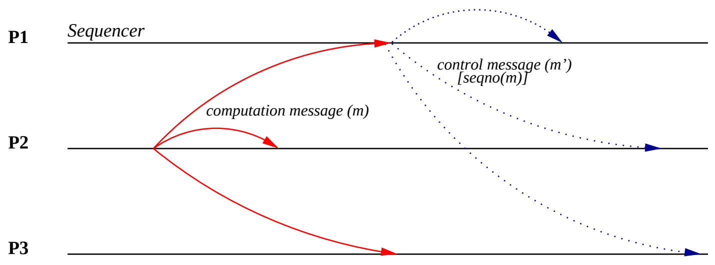


Figure 15: BB variant of fixed sequencer

As shown in the Fig. 15 process *P2* broadcast a *computation message m*. Upon delivery of m to *sequencer* process, sequencer assigns a sequence number and broadcast its sequence number through a *control message*(m'). Upon receipt of the control messages, a destination process deliver its computation message according to sequence numbers. In the following sections we present a formal analysis of *total order broadcast* with respect to the *Broadcast Broadcast*(*BB*) variant of the fixed sequencer approach.

In the following sections we outline incremental development of a system of total order broadcast. Our refinement chain consists of six levels. A brief outline of each level is given below.

L1 This consist of abstract model of total order broadcast. In this model, abstract total order is constructed when a message is delivered to a process for the first time. At all other processes a message is delivered in the total order.

L2 This is a refinement of abstract model which introduces *sequencer*. In this refinement we show that the total order is built by the *sequencer*.

L3 This is a very simple refinement giving more concrete specification of *Deliver* event. Through this refinement we illustrate that a total order can be built using the messages delivered to the sequencer.

L4 In this refinement we introduce the notion of *computation* messages. Global sequence numbers of the computation message are generated by the sequencer. The delivery of the messages is done based on the sequence numbers.

L5 In this refinement we introduce notion of *control* messages. We also introduce the relationship of each *computation* message with the *control* messages.

L6 A new event *Receive Control* is introduced. We illustrate that a process other than sequencer can deliver a *computation* message only if it has received *control* message for it.

5.1 Abstract Model of Total Order Broadcast

The abstract model of total order broadcast system is given in Fig. 16 and Fig. 17. The specification consists of four variables *sender*, *totalorder*, *tdeliver* and *delorder*. A brief description of the machine is given in the following steps.

```
MACHINE          TotalOrder
SETS             PROCESS; MESSAGE
VARIABLES        sender, totalorder, delorder, tdeliver
INVARIANT        sender ∈ MESSAGE ⇸ PROCESS
               ∧ totalorder ∈ MESSAGE ↔ MESSAGE
               ∧ delorder ∈  PROCESS ⇸ (MESSAGE ↔ MESSAGE)
               ∧ tdeliver ∈ PROCESS ↔ MESSAGE
INITIALISATION
                 sender := ∅     ||  totalorder :=∅
                 delorder := PROCESS × {∅}  ||  tdeliver := ∅
```

Figure 16: TotalOrder: Initial Part

The *sender* is a partial function from *MESSAGE* to *PROCESS*. The mapping $(m \mapsto p) \in sender$ indicates that message m was sent by process p. The variable *totalorder* is defined as a relation among the messages. A mapping of the form $(m1 \mapsto m2\ ) \in totalorder$ indicates that message *m1* is *totally ordered before m2*.

In order to represent the delivery order of messages at a process, variable *delorder* is used. A mapping $(m1 \mapsto m2) \in delorder(p)$ indicate that process p has delivered *m1* before *m2*. The variable *tdeliver* represent the messages delivered following a total order. A mapping of form $(p \mapsto m) \in tdeliver$ represents that a process p has delivered m following a *total order*.

The event *Broadcast* given in the Fig. 17 models the broadcast of a message. Similarly the event *Deliver* models the first ever delivery of a message to any process. The global *total order* on the messages is constructed when it delivered to a process for the first time. Later in the refinement we show that it is a role of a sequencer. The *TODeliver* models the delivery of the messages when a total order on the message has been constructed.

```
Broadcast (pp ∈ PROCESS , mm ∈ MESSAGE ) ≙
    WHEN        mm ∉ dom(sender)
    THEN        sender := sender ∪ {mm ↦ pp}
    END;

Deliver (pp ∈ PROCESS ,mm ∈ MESSAGE ) ≙
    WHEN        mm ∈ dom(sender)
              ∧ mm ∉ ran(tdeliver)
              ∧ ran(tdeliver) ⊆ tdeliver[{pp}]
    THEN     tdeliver := tdeliver ∪ {pp ↦ mm}
           || totalorder := totalorder ∪ ( ran(tdeliver) × {mm})
           || delorder(pp) := delorder(pp) ∪ ( tdeliver[{pp}] × {mm})
    END;

 TODeliver (pp ∈ PROCESS , mm ∈ MESSAGE) ≙
    WHEN        mm ∈ dom(sender)
              ∧ mm ∈ ran ( tdeliver )
              ∧ pp ↦ mm ∉ tdeliver
              ∧ ∀m.( m ∈ MESSAGE ∧ (m↦ mm) ∈ totalorder
                                        ⇒ (pp ↦ m) ∈ tdeliver)
    THEN     tdeliver := tdeliver ∪ {pp ↦ mm}
           || delorder(pp) := delorder(pp) ∪ ( deliver[{pp}] × {mm})
    END
```

Figure 17: TotalOrder: Events

Constructing a Total Order

The event *Deliver* models the delivery of a message (mm) at a process (pp) when it is delivered for the *first* time. The following guards of this event ensures that the message(mm) has not been delivered elsewhere and that each message delivered at any other process has also been delivered to this process(pp).

$$mm \notin ran(tdeliver)$$
$$ran(tdeliver) \subseteq tdeliver[\{pp\}]$$

Later in the refinement we show that this is a function of a designated process called *sequencer*. As a consequence of the occurrence of *Deliver* event, the message mm is delivered to the process pp and variable *totalorder* is updated by mappings in ($ran(tdeliver) \times mm$). This indicates that all messages delivered at any process in the system are *ordered* before mm. Similarly, the delivery order at the process is also updated such that all messages delivered at any process precedes mm. It can be noticed that the total order for a message is built when it is delivered to a process for the *first* time.

The event $TOdeliver(pp{,}mm)$ models the delivery of a message mm to a process pp respecting the *total order*. As the guard $mm \in ran(tdeliver)$ implies that the mm has been delivered to at least one process and it also implies that the total order on the message mm has also been constructed. Later in the refinement we show that process pp represents a process other than the *sequencer* process. The guard of the event ensure that message mm has already been delivered elsewhere and that all messages which precedes mm in abstract total order has also been delivered to pp.

5.2 Proof Obligations and Invariant Discovery

In this section we outline how the proof obligations generated by the B Tool guide us discovering new invariants. A full set of invariant properties of the model of total order broadcast system are given in Fig. 18 and 19 as first order predicates. The invariant *Inv-1* in Fig. 18 is the primary invariant which state total ordering property and the other invariants(*Inv 2-5*) are discovered when the proof obligations with respect to *Inv-1* are discharged. In the second step we add *Inv-6* in Fig. 19 as enforcement invariant which state transitivity on total order. Other invariants(*Inv 7-9*) are discovered when the proof obligations with respect to *Inv-6* are discharged. A process of finding new invariants is briefly outlined below.

	Invariants	**Required By**
*/*Inv-1*/*	$(m1 \mapsto m2) \in delorder(p)$ $\Rightarrow (m1 \mapsto m2) \in totalorder$	*Deliver, TOdeliver*
*/*Inv-2*/*	$(p \mapsto m1) \in tdeliver \wedge (p \mapsto m2) \notin tdeliver$ $\wedge\ m2 \in \mathrm{ran}(tdeliver)$ $\Rightarrow (m1 \mapsto m2) \in totalorder$	*TOdeliver*
*/*Inv-3*/*	$(p \mapsto m1) \in tdeliver \wedge (p \mapsto m2) \in tdeliver$ $\wedge\ (m2 \mapsto m1) \notin totalorder$ $\Rightarrow (m1 \mapsto m2) \in totalorder$	*Deliver, TOdeliver*
*/*Inv-4 */*	$(p1 \mapsto m1) \in tdeliver \wedge (p1 \mapsto m2) \notin tdeliver$ $\wedge\ (p2 \mapsto m1) \in tdeliver \wedge (p2 \mapsto m2) \in tdeliver$ $\Rightarrow (m1 \mapsto m2) \in totalorder$	*Deliver, TOdeliver*
*/*Inv-5*/*	$m \in MESSAGE \Rightarrow m \mapsto m \notin totalorder$	*Deliver, TOdeliver*

Figure 18: Invariants-I

The agreement and total order requirement imply that all correct process eventually deliver all messages in the same order [14]. Thus we add following invariant to our model as a primary invariant.

$$\forall(m1, m2, p).((m1 \mapsto m2) \in delorder(p) \Rightarrow (m1 \mapsto m2) \in totalorder)$$

When we add this invariant to our model two proof obligations were generated associated with the event *Deliver* and *TODeliver*. Proof obligation associated with the event *Deliver* was discharged using interactive prover, however the proof obligation associated with *TOdeliver* could not be discharged. Following is simplified form of proof obligation generated by interactive prover.

$$\begin{array}{l} TOdeliver(PO1) \\ \left[\begin{array}{l} p \mapsto m1 \in tdeliver \\ p \mapsto m2 \notin tdeliver \\ m2 \in ran(tdeliver) \\ \Rightarrow \\ m1 \mapsto m2 \in totalorder \end{array}\right] \end{array}$$

In order to discharge this proof obligation we add a invariant to our model given as *Inv-2* in Fig. 18. Addition of *Inv-2* was sufficient to discharge PO1, however a new proof obligation associated with *TODeliver* was generated due to addition of *Inv-2*. Following is the simplified form of the proof obligation.

$$TOdeliver(PO2)\left[\begin{array}{l} p \mapsto m1 \in tdeliver \\ p \mapsto m2 \in tdeliver \\ m2 \mapsto m1 \notin totalorder \\ \Rightarrow \\ m1 \mapsto m2 \in totalorder \end{array}\right]$$

In order to discharge the proof obligation we add another invariant *Inv-3* to our model. Addition of this invariant to the model further generate proof obligations. After three round of invariant strengthening we arrive at a set of invariant given in Fig. 18 which were sufficient to discharge all proof obligations.

	Invariants	**Required By**
/*Inv-6 */	$(m1 \mapsto m2) \in totalorder \wedge (m2 \mapsto m3) \in totalorder$ $\Rightarrow (m1 \mapsto m3) \in totalorder$	*Broadcast, Deliver TOdeliver*
/*Inv-7 */	$(m1 \mapsto m2) \in totalorder \wedge (p \mapsto m2) \in tdeliver$ $\Rightarrow (p \mapsto m1) \in tdeliver$	*Broadcast, Deliver TOdeliver*
/*Inv-8 */	$m \in (\,dom\,(totalorder) \cup ran(totalorder)\,)$ $\Rightarrow m \in ran(tdeliver)$	*Deliver*
/*Inv-9 */	$m \notin dom(\text{sender}) \Rightarrow m \notin dom(totalorder)$ $m \notin dom(\text{sender}) \Rightarrow m \notin ran(totalorder)$ $ran(tdeliver) \subseteq \text{dom(sender)}$	*Broadcast, Deliver TOdeliver*

Figure 19: Invariants-II

In the next step our goal was to verify that our model of Total Order also preserves transitive properties on the total ordering. In order to verify that *total order* is transitive, we add following as a enforcement invariant to the list of the invariants.

$$\begin{array}{l} (m1 \mapsto m2) \in totalorder \\ (m2 \mapsto m3) \in totalorder \\ \Rightarrow \\ (m1 \mapsto m3) \in totalorder \end{array}$$

Addition of this invariant generate proof obligations associated with the event *Broadcast*, *Deliver* and *TODeliver*. We are able to discharge proofs related with Broadcast event using interactive prover, however the following proof obligation associated with *Deliver* event could not be discharged by automatic prover.

$$Deliver(pp, mm)PO3\left[\begin{array}{l} (m1 \mapsto m2) \in totalorder \\ (p \mapsto m2) \in tdeliver \\ \Rightarrow \\ (p \mapsto m1) \in tdeliver \end{array}\right]$$

This property on the messages states that for two computation message *m1* and *m2* if *m1* is *totally ordered before m2* then for any process *p* who has delivered *m2* implies that it has also delivered *m1.* In order to discharge the this proof obligations we add *Inv-7* given in Fig. 19.

When we add this invariant to our model it generate further proof obligations associated with the events *Broadcast*, *Deliver* and *TOdeliver.* The proof obligation associated with *TOdeliver* is discharged using automatic prover. The simplified form of proof obligation associated with the events *BroadCast* which can not be discharged automatically is given below.

$$\begin{array}{l} BroadCast(pp, mm)PO4 \\ \left[\begin{array}{l} Inv7 \\ mm \notin dom(sender) \\ (pp \mapsto m2) \in tdeliver \\ (mm \mapsto m2) \in totalorder \\ m1 = mm \\ m2 \neq mm \\ \Rightarrow \\ (pp \mapsto mm) \in tdeliver \end{array}\right] \end{array}$$

It can be noticed that there is a contradiction in the hypothesis of this proof obligation i.e., the hypothesis *mm* $\notin$ *dom(sender)* and (*mm* $\mapsto$ *m2*) $\in$ *totalorder* can not be true simultaneously because of our assumption that a *totalorder* is built only when a message has been sent out. Similarly, the goal (*pp* $\mapsto$ *mm*) $\in$ *tdeliver* can not be proved under the hypothesis *mm* $\notin$ *dom(sender).* Thus we add following invariant(s) to our model given as Inv-8,9 in Fig. 19.

$$\begin{array}{l} \forall m\ .(\ m \in (\ dom(totalorder) \cup ran(totalorder)\ ) \Rightarrow m \in ran(tdeliver)) \\ \forall (m).(m \notin dom(sender)\ \Rightarrow m \notin ran(totalorder)) \\ \forall (m).(m \notin dom(sender)\ \Rightarrow m \notin dom(totalorder)) \\ ran(deliver) \subseteq dom(sender) \end{array}$$

Addition of these invariants were sufficient to discharge all proof obligations. Therefore after four iterations of invariant strengthening we arrive at a set of invariant that is sufficient to discharge all proof obligations generated due the addition of *Inv 6.* A full set of invariant are given in the Fig. 19.

5.3 First Refinement : Introducing Sequencer

In this refinement we introduce the notion of the sequencer. The refinement given in the Fig. 20 refines abstract model given in Fig. 16 and Fig. 17. The sequencer is defined as a constant for this model as *sequencer* $\in$ *PROCESS.* As shown in the refined specification of *Deliver* event given in Fig. 20, a message is first delivered to the sequencer process. It can be noticed that the the following guards in the abstract specification

$$\begin{array}{c} mm \notin ran(tdeliver) \\ ran(tdeliver) \subseteq tdeliver[pp] \end{array}$$

are replaced by following.

$$\begin{array}{c} pp = sequencer \\ (sequencer \mapsto mm) \notin tdeliver \end{array}$$

The replacement of the guards in the *Deliver* event generate new proof obligations. Using the same approach of invariant discovery, we arrived at a set of invariants which was sufficient to discharge all proof obligations. These invariants are given in Fig. 21. A brief description of these invariants are given in the following steps.

```
Broadcast (pp ∈ PROCESS , mm ∈ MESSAGE ) ≅
    WHEN        mm ∉ dom(sender)
    THEN        sender := sender ∪ {mm ↦ pp}
    END;

Deliver (pp ∈ PROCESS ,mm ∈ MESSAGE ) ≅
    WHEN        pp = sequencer
              ∧ mm ∈ dom(sender)
              ∧ (sequencer ↦ mm) ∉ tdeliver
    THEN        tdeliver := tdeliver ∪ {pp ↦ mm}
              || totalorder := totalorder ∪ ( ran(tdeliver) × {mm})
    END;

TODeliver (pp ∈ PROCESS , mm ∈ MESSAGE) ≅
    WHEN        pp ≠ sequencer
              ∧ mm ∈ dom(sender)
              ∧ mm ∈ ran ( tdeliver )
              ∧ pp ↦ mm ∉ tdeliver
              ∧ ∀m.( m ∈ MESSAGE ∧ (m↦ mm) ∈ totalorder
                                  ⇒ (pp ↦ m) ∈ tdeliver)
    THEN      tdeliver := tdeliver ∪ {pp ↦ mm}
    END
```

Figure 20: TotalOrder Refinement-I

– All messages delivered to any process in the system has also been delivered to the sequencer(*Inv-10*).

– A message not delivered to the sequencer is not delivered elsewhere(*Inv-11*).

– If a total order on any message m has been constructed then it must have been delivered to the sequencer(*Inv-12,13*).

Similarly it can be noticed a guard $pp \neq sequencer$ is added in the specifications of *TODeliver* event. Thus on occurrence of the event of *TODeliver*, a message mm is delivered to a process other than the sequencer.

	Invariants	**Required By**
/*Inv-10*/	$ran(tdeliver) \subseteq tdeliver[\{sequencer\}]$	*Deliver,TOdeliver*
/*Inv-11*/	$(sequencer \mapsto m) \notin tdeliver \Rightarrow m \notin ran(tdeliver)$	*Deliver*
/*Inv-12*/	$m \in dom(totalorder) \Rightarrow (sequencer \mapsto m) \in tdeliver$	*Deliver*
/*Inv-13*/	$m \in ran(totalorder) \Rightarrow (sequencer \mapsto m) \in tdeliver$	*Deliver*

Figure 21: TotalOrder Refinement-I : Invariants

5.4 Second Refinement : Refinement of Deliver event

This is a very simple refinement giving more concrete specification of *Deliver* event. Through this refinement we illustrate that a total order can be built using the messages delivered to the sequencer. As shown in the Fig. 20, a total order is generated as,

$$totalorder := totalorder \cup (ran(tdeliver) \times \{mm\})$$

It state that all messages delivered at any process are ordered before the new message *mm*. In the refined *Deliver* event the totalorder is constructed as below,

$$totalorder := totalorder \cup (tdeliver[\{sequencer\}] \times \{mm\})$$

It state that all messages delivered to the sequencer are ordered before the new message *mm*. The refined specifications of *Deliver* event are given in the Fig. 22.

```
Deliver (pp ∈ PROCESS ,mm ∈ MESSAGE ) ≙
  WHEN      pp = sequencer
          ∧ mm ∈ dom(sender)
          ∧ (sequencer ↦ mm) ∉ tdeliver
  THEN      tdeliver := tdeliver ∪ {pp ↦ mm}
          || totalorder := totalorder ∪ ( tdeliver[{sequencer}] × {mm})
  END;
```

Figure 22: TotalOrder Refinement-II : Refined Deliver Event

5.5 Third Refinement : Introducing the Sequence Numbers

In the third refinement we introduce the computation messages and the sequence numbers. The messages broadcast by the the processes which need to be delivered in the total order are called computation messages. In this intermediate refinement step, the sequence number to computation messages are assigned by the sequencer. This refinement introduces following new variables.

$$computation \subseteq MESSAGE$$
$$seqno \in computation \rightarrowtail\!\!\!\!\!+ \; Natural$$
$$counter \in Natural$$

Similarly, variable *seqno* is to assign sequence number to the computation messages. The *counter*, initialized with *zero*, is maintained by the sequencer process and incremented by *one* each time a control message is sent out by the *sequencer* process. It can be noted in the specification of *TODeliver* event that these message are delivered to the processes other than the sequencer in their sequence numbers. The refined model is given in the Fig. 23.

It can be noticed that following guard in the abstract *TODeliver*

$$\forall m.(m \in MESSAGE \wedge (m \mapsto mm) \in totalorder \Rightarrow (pp \mapsto m) \in tdeliver)$$

is replaced by

$$\forall m.(m \in MESSAGE \;\wedge\; seqno(m) < seqno(mm) \Rightarrow (pp \mapsto m) \in tdeliver)$$

The change of the guards in the *TODeliver* event generate new proof obligations. These proof obligations are discharged by adding following two invariant to the model.

```
Broadcast ($pp \in PROCESS$, $mm \in MESSAGE$) $\widehat{=}$
    WHEN    $mm \notin dom(sender)$
    THEN    $sender := sender \cup \{mm \mapsto pp\}$
          || $computation := computation \cup \{mm\}$
    END;

Deliver ($pp \in PROCESS$, $mm \in MESSAGE$) $\widehat{=}$
    WHEN    $pp = sequencer$
          $\wedge$ $mm \in dom(sender)$
          $\wedge$ $mm \in computation$
          $\wedge$ $(sequencer \mapsto mm) \notin tdeliver$
    THEN    $totalorder := totalorder \cup ( deliver[\{sequencer\}] \times \{mm\})$
          || $tdeliver := tdeliver \cup \{pp \mapsto mm\}$
          || $seqno := seqno \cup \{mm \mapsto counter\}$
          || $counter := counter + 1$
    END;

TODeliver ($pp \in PROCESS$, $mm \in MESSAGE$) $\widehat{=}$
    WHEN    $pp \neq sequencer$
          $\wedge$ $mm \in dom(sender)$
          $\wedge$ $mm \in ran ( tdeliver )$
          $\wedge$ $pp \mapsto mm \notin tdeliver$
          $\wedge$ $\forall m.( m \in MESSAGE \wedge ( seqno(m) < seqno(mm) )$
                              $\Rightarrow (pp \mapsto m) \in tdeliver)$
    THEN  $tdeliver := tdeliver \cup \{pp \mapsto mm\}$
    END
```

Figure 23: Refinement-III

$$\forall(m1, m2) \cdot (\ (m1 \mapsto m2) \in totalorder \Rightarrow seqno(m1) < seqno(m2) \ ) \quad (1)$$

$$\forall(m) \cdot (m \in computation \wedge m \in dom(seqno) \Rightarrow sequencer \mapsto m \in tdeliver) \quad (2)$$

5.6 Fourth Refinement : Introducing Control Messages

In this refinement we introduces control messages. The messages are classified as either a *computation* message or a *control* message. A *computation* message is one which can be sent by any process and that need to be delivered following a total order. For each computation message there is an associated *control* message. The control messages are sent by the sequencer process once it delivers a computation message. We have illustrated this protocol in the Fig. 15.

In this refinement of total order broadcast,a process broadcasts a computation message *mm* to all processes including the *sequencer.* Upon delivery of this message, the sequencer assigns it a sequence number and broadcast its *control* message. All process except the *sequencer* deliver the corresponding computation messages in the order of the *sequence numbers.* This refinement also consists of following new state variables typed as follows,

```
Broadcast ($pp \in PROCESS$ , $mm \in MESSAGE$ ) $\widehat{=}$
    WHEN    $mm \notin dom(sender)$
    THEN    $sender := sender \cup \{mm \mapsto pp\}$
          || $computation := computation \cup \{mm\}$
    END;
Deliver ($pp \in PROCESS$ ,$mm \in MESSAGE$,$mc \in MESSAGE$ ) $\widehat{=}$
    WHEN    $pp = sequencer$
          $\wedge$ $mm \in dom(sender)$
          $\wedge$ $mm \in computation$
          $\wedge$ $(sequencer \mapsto mm) \notin tdeliver$
          $\wedge$ $mc \notin dom(messcontrol)$
          $\wedge$ $mm \notin ran(messcontrol)$
    THEN    $totalorder := totalorder \cup ( tdeliver[\{sequencer\}] \times \{mm\})$
          || $tdeliver := tdeliver \cup \{pp \mapsto mm\}$
          || $control := control \cup \{mc\}$
          || $messcontrol := messcontrol \cup \{mc \mapsto mm\}$
          || $seqno := seqno \cup \{mm \mapsto counter\}$
          || $counter := counter + 1$
    END;
TODeliver ($pp \in PROCESS$ , $mm \in MESSAGE$) $\widehat{=}$
    WHEN    $pp \neq sequencer$
          $\wedge$ $mm \in dom(sender)$
          $\wedge$ $mm \in ran ( messcontrol )$
          $\wedge$ $pp \mapsto mm \notin tdeliver$
          $\wedge$ $\forall m.( m \in MESSAGE \wedge ( seqno(m) < seqno(mm) )$
                                $\Rightarrow (pp \mapsto m) \in tdeliver)$
    THEN  $tdeliver := tdeliver \cup \{pp \mapsto mm\}$
    END
```

Figure 24: Refinement-IV

$$control \subseteq MESSAGE$$
$$messcontrol \in control \rightarrowtail\!\!\!\!\!\!\rightarrow computation$$

The variables *control* and *computation* are used to cast a message as either a computation or a control message. The set *control* contains the control messages sent by the sequencer. The variable *messcontrol* is a partial injective function which defines relationship among a control message and its computation message. A mapping $(m1 \mapsto m2) \in messcontrol$ indicate that message *m1* is the *control message* related to the *computation message m2*. Since *messcontrol* is defined as a partial injective function, it also implies that there can only be a one control message for each computation message and vice-versa. The set *ran(messcontrol)* contains the computation messages for which control messages has been sent by the sequencer. The refined model is given in the Fig. 24.

The change in the guards in *Deliver* and *TOdeliver* generate some proof obligations which are discharged by adding following invariant to the model.

$$ran(messcontrol) = ran(tdeliver) \tag{3}$$

$$ran(messcontrol) \subseteq computation \tag{4}$$

5.7 Fifth Refinement : Introducing Receive Control Event

A new event *ReceiveControl* is introduced in this refinement. This event, which refines *skip* in the abstraction, models receiving of control messages at a processes. A new variable *receive* is also introduced in this refinement as $receive \in$ PROCESS $\leftrightarrow$ *control*. A mapping $p \mapsto m \in receive$ indicate that process p has received control message m. The specifications of the refined events are given in Fig. 25, 26.

```
Broadcast ($pp \in PROCESS$ , $mm \in MESSAGE$ ) $\widehat{=}$
    WHEN      $mm \notin dom(sender)$
    THEN      $sender := sender \cup \{mm \mapsto pp\}$
            || $computation := computation \cup \{mm\}$
    END;

Deliver ($pp \in PROCESS$ ,$mm \in MESSAGE$,$mc \in MESSAGE$ ) $\widehat{=}$
    WHEN      $pp = sequencer$
          $\wedge$ $mm \in dom(sender)$
          $\wedge$ $mm \in computation$
          $\wedge$ $mm \notin ran(messcontrol)$
          $\wedge$ $mc \notin dom(messcontrol)$
          $\wedge$ $(sequencer \mapsto mm) \notin tdeliver$
    THEN      $totalorder := totalorder \cup ( tdeliver[\{pp\}] \times \{mm\})$
          || $tdeliver := tdeliver \cup \{pp \mapsto mm\}$
          || $messcontrol := messcontrol \cup \{mc \mapsto mm\}$
          || $control := control \cup \{mc\}$
          || $seqno := seqno \cup \{mm \mapsto counter\}$
          || $counter := counter + 1$
    END;
```

Figure 25: Refinement-V : Part-1

As shown in the Fig. 25, the event *BroadCast*(*pp*,*mm*) models the broadcast of a *computation* message. The *Deliver* models the event of sending a control message once a *computation message* is delivered at the *sequencer*. It can be noticed that a *sequencer* assigns a *sequence number* to each computation message using a counter. Variable *messcontrol* is also updated to contain the relationship of a control and a computation message. The entry $mc \in control$, indicate that message *mc* is a control message sent by the sequencer. The sequence number of the associated computation message is sent through the control message and a recipient of a control message delivers the computation message according the sequence numbers. The new event *ReceiveControl* models the recipe of the control message at processes. The variable *receive* is updated when a control message is received at a process. The event *TOdeliver* models event of delivery of a *computation* message to a process. As shown in the Fig. 26, the guard $mm \in ran(messcontrol)$ is replaced by the following,

$$(pp \mapsto messcontrol^{-1}(mm)) \ \in \ receive$$

This guard of the *TOdeliver* event ensures that a process *pp* delivers a computation message *mm* only when its corresponding control message has been received by the process *pp*. The change in the guards generate the proof obligations associated with the event *TOdeliver*. In order to discharge these proof obligation we add following to the list of invariants.

$$\forall m \cdot (m \in computation \wedge messcontrol^{-1}(m) \in receive \Rightarrow m \in ran(messcontrol))$$

```
**ReceiveControl** $(pp \in PROCESS, mc \in MESSAGE) \cong$
    WHEN  $mc \in control$
          $\wedge\ (pp \mapsto mc) \notin receive$
    THEN  $receive := receive \cup \{pp \mapsto mc\}$
    END

**TODeliver** $(pp \in PROCESS, mm \in MESSAGE) \cong$
    WHEN  $pp \neq sequencer$
          $\wedge\ mm \in dom(sender)$
          $\wedge\ mm \in computation$
          $\wedge\ (pp \mapsto mm) \notin tdeliver$
          $\wedge\ (pp \mapsto messcontrol^{-1}(mm)) \in receive$
          $\wedge\ \forall m.(m \in MESSAGE \wedge (seqno(m) < seqno(mm)$
                                      $\Rightarrow (pp \mapsto m) \in tdeliver)$
    THEN  $tdeliver := tdeliver \cup \{pp \mapsto mm\}$
    END
```

Figure 26: Refinement-V : Part-2

Proof Statistics

The over all proof statistics for the development of a system of total order broadcast is given in Table 2. Approximately seventy five percent of the proofs were discharged by the automatic prover, the rest were discharged by using interactive prover of B Tool.

Machine	Total POs	Completely Automatic	Required Interaction
Abstract Model	48	29	19
Refinement1	19	16	03
Refinement2	2	2	00
Refinement3	18	14	04
Refinement4	15	14	01
Refinement5	04	04	00
Total	106	79	27

Table 2: Proof Statistics- Total Order Broadcast

6 Total Causal Order Broadcast

We have presented B specifications of causal order broadcast system and the total order broadcast system as separate development in the sections 4 and 5. In this section we present B specification of an execution model of a system which respect both total and a causal order on the computation messages. A process is said to *codeliver* a message when it is delivered following a causal order, similarly a process is said to *todeliver* a message when it is delivered following a total order. In this model we use a fixed sequencer algorithm to implement total order. The protocol works as follows. A process first broadcasts a computation message after building a causal order. This computation message is *codelivered* to all processes including the *sequencer*. As discussed in the section 5, the sequencer is an specific process which builds the total order on the computation messages. Upon *codelivery* of the computation message at the sequencer process, sequencer assigns computation message a sequence number and further broadcast its sequence number through control message. All messages inclusive of control message in our model are *codelivered*. Since a broadcast of a *computation message* causally precedes the broadcast of its control message, each process *codelivers* computation message before it delivers its control message. Upon *codelivery* of a control message, a process *todelivers* the computation message in the order of sequence numbers. As shown in Fig. 27, when two process *P2* and *P3* broadcast computation messages *M1* and *M2*, they are first *codelivered* to all processes including the sequencer. The sequencer builds a total order on the *computation messages* in the order they were *codelivered* to the sequencer and broadcast their sequence number through their respective *control messages*. When a process *codelivers* the control message, it then *todeliver* the corresponding computation message. Thus each process *codelivers* the *control message* respecting the causality of their respective *computation message*.

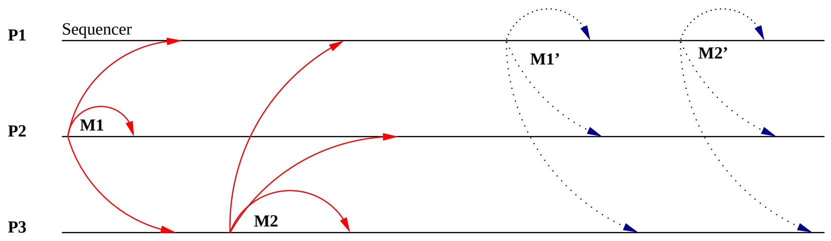


Figure 27: Execution Model of Total Causal Order

The incremental development of system of Total Causal Order broadcast is developed in three levels. These levels are outlined below.

L1 This level consists of abstract model of causally and totally ordered broadcast.

L2 In this refinement step we replace abstract variables *causalorder* and *totalorder* with the vector clock rules and sequence numbers respectively.

L3 In this refinement we show that sequence numbers are redundant and they can be replaced by the vector clock rules. We also present the gluing invariants relating abstract *causalorder*, *totalorder*, vector clocks and sequence numbers.

6.1 Abstract Model of Total Causal Order Broadcast

The initial part of abstract model of this system is given as *TotalCausalOrder* in Fig. 28 as a B Machine. The specifications of the events of the machine are given in Fig. 29 and Fig. 30. As shown in the Fig. 28, *sequencer* is defined as a constant. Variable *sender* is used to represent the messages broadcasted by a process.

The variable *cdeliver* represents the messages *codelivered* to the processes following a causal order. Similarly, variable *tdeliver* represent the messages *todelivered* to the processes following a total order. The variable *messcontrol* is a partial injective function which defines relationship between a *computation message* and its *control message*. A mapping $(m1 \mapsto m2) \in messcontrol$ indicate that message *m1* is the *control message* related to the *computation message m2*. The set *ran(messcontrol)* contains the computation messages for which control messages has been sent by the sequencer.

In order to represent the causally ordered delivery of the messages at a process, variable *cdeloder* is used. A mapping of the form $(m1 \mapsto m2) \in cdelorder(p)$ indicate that the process *p* has *codelivered m1* before *m2*. Similarly, a mapping $m1 \mapsto m2)$

```
MACHINE        TotalCausalOrder
CONSTANTS      sequencer
PROPERTIES     $sequencer \in PROCESS$
SETS           PROCESS; MESSAGE;
               $MTYPE = \{Computation , Control\}$
VARIABLES      sender , cdeliver , tdeliver , mtype ,
               messcontrol , causalorder , totalorder,
               cdelorder,tdelorder
INVARIANT      $sender \in MESSAGE \rightarrow PROCESS$
             $\wedge$ $cdeliver \in PROCESS \leftrightarrow MESSAGE$
             $\wedge$ $tdeliver \in PROCESS \leftrightarrow MESSAGE$
             $\wedge$ $mtype \in MESSAGE \rightarrow MTYPE$
             $\wedge$ $messcontrol \in MESSAGE \rightarrowtail MESSAGE$
             $\wedge$ $causalorder \in MESSAGE \leftrightarrow MESSAGE$
             $\wedge$ $totalorder \in MESSAGE \leftrightarrow MESSAGE$
             $\wedge$ $cdelorder \in PROCESS \rightarrow (MESSAGE \leftrightarrow MESSAGE)$
             $\wedge$ $tdelorder \in PROCESS \rightarrow (MESSAGE \leftrightarrow MESSAGE)$

INITIALISATION
               $sender := \varnothing$ || $cdeliver := \varnothing$ || $tdeliver := \varnothing$ ||
               $mtype := \varnothing$ || $messcontrol := \varnothing$ || $causalorder := \varnothing$ ||
               $totalorder := \varnothing$ || $cdelorder := PROCESS \times \{\varnothing\}$ ||
               $tdelorder := PROCESS \times \{\varnothing\}$
```

Figure 28: TotalCausalOrder: Initial Part

```
BroadCast ($pp \in PROCESS , mm \in MESSAGE$) $\cong$
    WHEN   $mm \notin dom(sender)$
    THEN   $sender := sender \cup \{mm \mapsto pp\}$
       ||  $causalorder := causalorder \cup ( (sender^{-1}[\{pp\}] \times \{mm\})$
                                     $\cup ( cdeliver[\{pp\}] \times \{mm\}))$
       ||  $mtype(mm) := Computation$
    END;

CausalDeliver($pp \in PROCESS , mm \in MESSAGE$) $\cong$
    WHEN   $mm \in dom(sender)$
       $\wedge$  $(pp \mapsto mm) \notin cdeliver$
       $\wedge$  $\forall m.( m \in MESSAGE \wedge (m \mapsto mm) \in causalorder$
                               $\Rightarrow (pp \mapsto m) \in cdeliver)$
    THEN   $cdeliver := cdeliver \cup \{pp \mapsto mm\}$
       ||  $cdelorder(pp) := cdeloder(pp) \cup (cdeliver[\{pp\}] \times \{mm\})$
    END;
```

Figure 29: TotalCausalOrder: Events-I

$\in$ *tdelorder(p)* indicate that the process p has *todelivered m1* before *m2*. It may be noted that a message may have been *codelivered* at a process but still waiting for it to be *todelivered*.

```
SendControl ($pp \in PROCESS, mm \in MESSAGE, mc \in MESSAGE$) $\cong$
  WHEN    $pp = sequencer$
          $\wedge\ mc \notin dom(sender)$
          $\wedge\ mm \notin ran(messcontrol)$
          $\wedge\ mtype(mm) = Computation$
          $\wedge\ (pp \mapsto mm) \in cdeliver$
          $\wedge\ \forall m.\ (\ m \in MESSAGE\ \wedge (m \mapsto mm) \in causalorder$
                      $\Rightarrow\ m \in ran\ (messcontrol))$
  THEN    $causalorder := causalorder\ \cup (\ (sender^{-1}[\{sequencer\}] \times \{mc\})$
                      $\cup (\ cdeliver[\{sequencer\}] \times \{mc\}))$
       || $sender := sender \cup \{mc \mapsto sequencer\}$
       || $mtype(mc) := Control$
       || $messcontrol := messcontrol \cup \{mc \mapsto mm\}$
       || $LET\ m\ BE\ m = ran(messcontrol)$
          $IN\ totalorder := totalorder \cup (\ m \times \{mm\})\ END$
  END;

TODeliver ($pp \in PROCESS, mc \in MESSAGE$) $\cong$
  WHEN  $mc \in dom(sender)$
     $\wedge$  $mtype(mc)=Control$
     $\wedge$  $(pp \mapsto mc) \in cdeliver$
     $\wedge$  $(pp \mapsto messcontrol(mc)) \in cdeliver$
     $\wedge$  $(pp \mapsto messcontrol(mc)) \notin tdeliver$
     $\wedge\ \forall m.(\ m \in MESSAGE \wedge (m \mapsto messcontrol(mc)) \in totalorder$
                $\Rightarrow (pp \mapsto m) \in tdeliver)$
  THEN   $tdeliver := tdeliver \cup\ \{pp \mapsto messcontrol(mc)\}$
  || $tdelorder(pp) := tdeloder(pp) \cup (tdeliver[\{pp\}] \times \{messcontrol(mc)\})$
  END
```

Figure 30: TotalCausalOrder: Event-II

The *Broadcast* event given in the Fig. 29 models the broadcast of a *computation* message. It can be noticed that a *causal order* is built by the sender process while broadcasting a *computation* message. The event *CausalDeliver* models the event of causally ordered delivery of message to a process. The guards of the *CausalDeliver* also ensures that a message is *codelivered* only once. The following guards of the *CausalDeliver* event ensure that a process *pp* causally *codelivers* a message *mm* only if it has *codelivered* all messages which causally precedes *mm.*

$$\forall m.((m \mapsto mm)\ \in\ causalorder \Rightarrow (pp \mapsto m) \in cdeliver)$$

Upon delivery of a message *mm* in causal order the variable *cdelorder* is also updated so that all messages *codelivered* to process *pp* are ordered before *mm.*

The specifications of the events *SendControl* and *TOdeliver* are given in Fig. 30. The *SendControl* is an event of sending a control message once a computation message is *codelivered* at the *sequencer.* The following guard of this event ensure that a control message(mc) for a computation message(mm) is broadcasted only when it has already broadcasted control messages for the computation messages which *causally precedes mm.*

$$\forall m.((m \mapsto mm) \in causalorder \Rightarrow m \in ran(messcontrol))$$

The set *ran(messcontrol)* contains the computation messages for which control messages has been sent by the sequencer. In the operations of event *SendControl*, it can be noticed that the sequencer also builds the causal order on the control messages and the variable *messcontrol* is updated by adding a corresponding mapping. A total order for the computation messages *mm* is also build by the sequencer by updating abstract variable *totaloder* as :

$$totalorder := totalorder \cup (m \times \{mm\})$$

where $m = ran(messcontrol)$. This implies that all computation messages, for which sequencer has already sent out control messages, are now totally ordered before *mm*.

The event *TOdeliver* event models of totally ordered delivery of a computation message to a process. This event is activated when a process *pp codelivers* a control message *mc*. The guard of the event ensures that on *codelivery* of a control message *mc* by a process *pp*, it delivers a computation message in total order corresponding to the control message *mc* if it has already delivered all computation messages which are *totally ordered before* computation message defined as *messcontrol(mc)*. It may be noted that *messcontrol(mc)* represents a computation message corresponding to the control message *mc*. Upon *todelivery* of a message *mm* the variable *tdelorder* is also updated so that all messages *todelivered* to the process *pp* are ordered before *mm*.

6.2 Invariant Properties of Total Causal Order

Some of the important invariant properties of our model of *total causal order* are given in Fig. 31 as first order predicates. The codes for the events are given in the Table 3. A brief description of these properties is given below.

- If message *m1* causally precedes *m2* and a process *p* has *codelivered m2* then delivery order at the process p must have been *m1* followed by *m2 (Inv-1)*.
- For two messages *m1* and *m2* where *m1* is *todelivered* before *m2* at a process *p* ($m1 \mapsto m2 \in delorder(p)$) then *m1* precedes *m2* in abstract total order *(Inv-2)*.
- A message is first *codelivered* to a process then it is *todelivered* to the process. This invariant state that a message delivered in a total order is also in a causal order *(Inv-3)*.
- The *transitivity* property on *causal precedence* relationship on the messages holds on our model of causally and totally ordered broadcast*(Inv-4)*.
- For two messages *m1* and *m2*, if *m1* causally precedes *m2* and process *p* has *codelivered* the message *m2* then *p* has also *codelivered* the message *m1 (Inv-5)*.
- The *transitivity* property on *Total Order* relationship on the messages also holds on our model of causally and totally ordered broadcast*(Inv-6)* .
- For two messages *m1* and *m2*, if *m1* precedes *m2* in *total order* and process *p* has *todelivered* the message *m2* then *p* has also *todelivered m1 (Inv-7)*.
- For any two messages *m1* and *m2* whose control message has been sent out i.e., *m1*,*m2* ∈ *ran(messcontrol)* and that *m1 causally precedes m2* then a *total order* also exist among them i.e. *m1* is *totally ordered before m2*. It can be noticed that that message type of each message in the set *ran(messcontrol)* is computation *(Inv-8)* .

	Invariants	**Required By**
*/*Inv-1*/*	$(m1 \mapsto m2) \in causalorder \wedge (p \mapsto m2) \in cdeliver \Rightarrow (m1 \mapsto m2) \in cdeloder(p)$	*BC,CD,SC*
*/*Inv-2*/*	$(m1 \mapsto m2) \in tdelorder(p) \Rightarrow (m1 \mapsto m2) \in totalorder$	*SC,TOD*
*/*Inv-3*/*	$(p \mapsto m) \in tdeliver \Rightarrow (p \mapsto m) \in cdeliver$	
*/*Inv-4*/*	$(m1 \mapsto m2) \in causalorder \wedge (m2 \mapsto m3) \in causalorder \Rightarrow (m1 \mapsto m3) \in causalorder$	*BC,CD,SC*
*/*Inv-5*/*	$(m1 \mapsto m2) \in causalorder \wedge (p \mapsto m2) \in cdeliver \Rightarrow (p \mapsto m1) \in cdeliver$	*BC,CD, SC,TOD*
*/*Inv-6*/*	$(m1 \mapsto m2) \in totalorder \wedge (m2 \mapsto m3) \in totalorder \Rightarrow (m1 \mapsto m3) \in totalorder$	*SC,TOD*
*/*Inv-7*/*	$(m1 \mapsto m2) \in totalorder \wedge (p \mapsto m2) \in tdeliver \Rightarrow (p \mapsto m1) \in tdeliver$	*SC,TOD*
*/*Inv-8*/*	$m1 = ran(messcontrol) \wedge m2 = ran(messcontrol) \wedge (m1 \mapsto m2) \in causalorder \Rightarrow (m1 \mapsto m2) \in totalorder$	*BC,SC, TOD*
*/*Inv-9*/*	$mtype(\mathrm{m1})=Computation \wedge mtype(m2)=Computation \wedge (m1 \mapsto m2) \in causalorder \wedge m2 \in ran(messcontrol) \Rightarrow m1 \in ran(messcontrol)$	*BC,SC, TOD*
*/*Inv-10*/*	$m \in ran(messcontrol) \Rightarrow (sequencer \mapsto m) \in cdeliver$	*CD,SC*

Figure 31: Invariants-I

- For any two computation messages *m1* and *m2* where *m1* causally precedes *m2* and the control messages for *m2* have been sent out implies that the control message for *m1* have also been sent*(Inv-9)*.
- Each message whose control message have been sent, have also been *codelivered* at the sequencer*(Inv-10)*.

BC	BroadCast	CD	CausalDeliver
SC	SendControl	TOD	TODeliver

Table 3: Events Code

6.3 First Refinement : Introducing Vector Clocks

The first refinement of the machine *TotalCausalOrder* through vector clocks and sequence numbers is shown in the Fig. 32, 33 and Fig. 34. The variables *VTP*, *VTM* and *seqno* are introduced in this refinement representing vector clock of a process, vector timestamp of a message and the sequence numbers respectively.

It can be noticed that operation of events (*Broadcast*, *CausalDeliver* and *SendControl*) involving abstract variable *causalorder* are replaced by the vector rules. The operations of events *SendControl* and *TOdeliver* involving abstract variable *totalorder* are replaced by sequence numbers(*seqno*).

```
REFINEMENT        Vector
REFINES           TotalCausalOrder

VARIABLES         sender, cdeliver, mtype, messcontrol,
                  tdeliver, seqno, counter, VTP,VTM

INVARIANT         $VTP \in PROCESS \rightarrow (PROCESS \rightarrow NAT)$
                $\wedge\ VTM \in MESSAGE \rightarrow (PROCESS \rightarrow NAT)$
                $\wedge\ seqno \in MESSAGE \nrightarrow NATURAL$
                $\wedge\ counter \in NATURAL$

INITIALISATION    $VTP := PROCESS \times \{PROCESS \times \{0\}\}$
               $\parallel\ VTM := MESSAGE \times \{PROCESS \times \{0\}\}$
               $\parallel sender := \varnothing \quad \parallel cdeliver := \varnothing \quad \parallel seqno := \varnothing$
               $\parallel counter := 0 \quad \parallel mtype := \varnothing$
               $\parallel messcontrol := \varnothing \quad \parallel tdeliver := \varnothing$
```

Figure 32: First Refinement- Part I

```
Broadcast($pp \in PROCESS$ , $mm \in MESSAGE$) $\widehat{=}$
  WHEN  $mm \notin dom(sender)$
  THEN  LET $nVTP$ BE $nVTP = VTP(pp) \lhd\!\!- \{ pp \mapsto VTP(pp)(pp)+1\}$
          IN  $VTM(mm) := nVTP \quad \parallel VTP(pp) := nVTP$  END
        $\parallel sender := sender \cup \{mm \mapsto pp\}$
        $\parallel mtype(mm) := Computation$
  END ;
```

```
CausalDeliver ($pp \in PROCESS$ , $mm \in MESSAGE$) $\widehat{=}$
   WHEN  $mm \in dom(sender)$
        $\wedge\ (pp \mapsto mm) \notin cdeliver$
        $\wedge\ \forall p.( p \in PROCESS \ \wedge p \neq sender(mm)$
                    $\Rightarrow VTP(pp)(p) \geq VTM(mm)(p))$
        $\wedge\ VTP(pp)(sender(mm)) = VTM(mm)(sender(mm))-1$
   THEN
         $cdeliver := cdeliver \cup \{pp \mapsto mm\}$
       $\parallel\ VTP(pp) := VTP(pp) \lhd\!\!-$
         $(\{q \mid q \in PROCESS \wedge VTP(pp)(q) < VTM(mm)(q)\} \lhd VTM(mm))$
   END;
```

Figure 33: First Refinement- Part II

The vector time of a process is represented by a variable *VTP*. The timestamp

```
**SendControl** $(pp \in PROCESS,\ mm \in MESSAGE,\ mc \in MESSAGE) \widehat{=}$
  WHEN $pp = sequencer$
        $\wedge\ mc \notin dom(sender)$
        $\wedge\ mm \notin ran(messcontrol)$
        $\wedge\ mtype(mm) = Computation$
        $\wedge\ (pp \mapsto mm) \in cdeliver$
        $\wedge\ \forall (m,p) \cdot (\ p \in PROCESS \wedge m \in MESSAGE \wedge mtype(m) = Computation$
                    $\wedge\ VTM(m)(p) \leq VTM(mm)(p) \ \Rightarrow\ m \in ran(messcontrol))$
  THEN $mtype(mc) := Control$
     || $messcontrol := messcontrol \cup \{mc \mapsto mm\}$
     || LET $nVTP$ BE $nVTP = VTP(pp) \Leftarrow \{\ pp \mapsto VTP(pp)(pp)+1\}$
        IN $VTM(mc) := nVTP \ \|\ VTP(pp) := nVTP$ END
     || $sender := sender \cup \{mc \mapsto pp\}$
     || LET $ncount$ BE $ncount = counter + 1$
        IN $counter := ncount \ \|\ seqno(mc) := ncount$ END
  END;

**TODeliver** $(pp \in PROCESS,\ mc \in MESSAGE) \widehat{=}$
  WHEN $mc \in dom(sender)$
        $\wedge\ mtype(mc) = Control$
        $\wedge\ (pp \mapsto messcontrol(mc)) \notin tdeliver$
        $\wedge\ \forall m.(\ m \in MESSAGE \wedge (seqno(messcontrol^{-1}(m)) < seqno(mc))$
                                        $\Rightarrow (pp \mapsto m) \in tdeliver$
  THEN $tdeliver := tdeliver \cup \{pp \mapsto messcontrol(mc)\}$
  END
```

Figure 34: First Refinement - Part III

of a message is represented by a variable *VTM*. *VTP* and *VTM* are defined as functions. These functions maps the processes and messages to a vector of integers. Vector timestamp of each process is initialized with value '0'. Two new variables *seqno* and *counter* are also introduced in this refinement. The function *seqno* maps a message to a number called *sequence number*. The *counter* is a variable maintained by the sequencer processes and it is updated each time a control message is sent by the sequencer.

The events *Broadcast*(Fig. 33) and *SendControl*(Fig. 34) are events of sending a message. In both of the events, sender process *pp* increments its own clock value *VTP(pp)(pp)* by one. Recall that *VTP(pp)(pp)* represents the number of messages sent by process *pp*. The modified vector timestamp of process is also assigned to message *mm* giving vector timestamp of message *mm*.

The *CausalDeliver*(Fig. 33) events models causally ordered delivery of a message *mm* at process *pp*. The following guard of the event involving abstract order,

$$\forall m.((m \mapsto mm) \in causalorder \Rightarrow (pp \mapsto m \in cdeliver)$$

is replaced by following vector rules in the refinement.

(1) $\forall p.(p \in PROCESS \wedge\ p \neq sender(mm) \Rightarrow VTP(pp)(p) \geq VTM(mm)(p))$
(2) $VTP(pp)(sender(mm)) = VTM(mm)(sender(mm)) - 1$

The first condition state that the vector timestamp of a recipient process *pp* and message *mm* are compared to ensure that all messages received by sender of message before sending it, are also received at the recipient process. The second condition state that process *pp* has received all but one message from the sender of the message *mm* . An operation updating vector clock of recipient process *pp* is also shown in the specification of *CausalDeliver* event.

The variable *seqno* is used for building a total order on the computation messages. In the refined specification of event *SendControl*, it can be noticed that operation involving abstract *totalorder* is replaced by an operation containing variable *seqno* and *counter*. The counter is incremented each time a control message is sent and it is assigned to the control messages.

The guards of the event *TOdeliver* are strengthened in this refinement. It can be noticed that following guard of the event *TOdeliver* involving abstract *totalorder*

$$\forall m.((m \mapsto messcontrol(mc) \in totalorder \Rightarrow (pp \mapsto m) \in tdeliver)$$

is replaced by a following guard involving sequence numbers. It state that the process has *todelivered* all computation messages *m* where the sequence number of corresponding control message is less than the sequence number of *mc*.

$$\forall m.(\ seqno(messcontrol^{-1}(m)) < seqno(mc) \Rightarrow (pp \mapsto m) \in tdeliver)$$

6.4 Gluing Invariants(Refinement-1)

The invariant showing the relationship of abstract *order* and *totalorder* with the *vector rules* and *sequence numbers* is given in the Fig. 35. The code for the events are given in Table 3. A process of construction of gluing invariant is given in section 6.5. A brief description of these properties are given below.

- If the vector time of process *P* is equal or more than vector time stamp of any sent message *M* then *P* must have *codelivered* the message *M* (*Inv-11*).
- For any two messages *m1* and *m2* where *m1* causally precedes *m2*, the vector time stamp of *m1* is less than vector time stamp of *m2*(*Inv-12*)
- Since *VTP(p)(p)* represent total number of message sent by process *p* and *VTM(m)(p)* represent number of message received by sender of *m* from process *p* before sending *m* , the number of messages sent by process *p* will be

	Invariants	**Required By**
*/*Inv-11*/*	$m \in dom(sender) \wedge VTP(p1)(p2) \geq VTM(m)(p2) \Rightarrow (p1 \mapsto m) \in cdeliver\ )$	*BC,CD,SC*
*/*Inv-12*/*	$(m1 \mapsto m2) \in causalorder \Rightarrow VTM\ (m1)(p) \leq VTM(m2)(p))$	*BC,CD*
*/*Inv-13*/*	$m \in dom(sender) \Rightarrow VTM(m)(p) \leq VTP(p)(p)\ )$	*BC,CD*
*/*Inv-14*/*	$VTM\ (m)(p)=0 \Rightarrow m \notin (\ dom(causalorder) \cup ran(causalorder))$	*BC,CD*
*/*Inv-15*/*	$messcontrol(m1) \mapsto messcontrol(m2) \in totalorder \Rightarrow seqno(m1) < seqno(m2)\ )$	*SC,TOD*

Figure 35: Gluing Invariants-IV

greater than or equal to the number of messages received by *sender(m)* from *p* (*Inv-13*).

- A message whose time stamp is a vector of zero's implies that it is not causally ordered(*Inv-14*). It may be noted that due to invariants $dom(order) \subseteq dom(sender)$ and $ran(order) \subseteq dom(sender)$ (given in Fig. **??**), it is also implied that these massages have not been sent.

- If computation messages corresponding to the control messages *m1* and *m2* are in *totalorder* then sequence number of *m1* is less then the sequence number of *m2* (*Inv-15*).

6.5 Constructing Gluing Invariants

In this section we briefly outline how the gluing invariants given in Fig 35 are constructed.

Relationship of abstract causal order and vector clock rules

The replacement of the guards and operations involving variable *causalorder* in the abstract model by the equivalent rules of *vector clock* generate several proof obligations due to refinement checking. Initially, the only proof obligation that can not be proved is given below. It involve relationship between *causalorder* and vector timestamp of message generated by the event *CausalDeliver.*

$$
\begin{array}{l}
CausalDeliver(pp, mm)PO1 \\
\left[\begin{array}{l}
mm \in dom(sender) \land \\
(pp \mapsto mm) \notin cdeliver \land \\
\forall p.(p \in PROCESS \Rightarrow VTM(mm)(p) \geq VTP(pp)(p) \land \\
VTP(pp)(sender(mm)) = VTM(mm)(sender(mm)) - 1 \land \\
m \in MESSAGE \\
m \mapsto mm \in causalorder \\
\Rightarrow \\
(pp \mapsto m) \in cdeliver
\end{array}\right]
\end{array}
$$

In this proof obligation it can be noticed that a message *m* causally precedes *mm* i.e.,(m $\mapsto$ mm) $\in$ *causalorder* and process *pp* has not *codelivered mm.* According to vector clock rules, *pp* can *codeliver mm* only when it has *codelivered* all messages inclusive of *m* which causally precedes *mm.* If a process pp has *codelivered* all but one message from sender of *mm* then following must be hold,

$$VTP(pp)(sender(mm)) = VTM(mm)(sender(mm)) - 1$$

Similarly, if a process *pp* has *codelivered* all messages sent by sender of *mm* before sending *mm* and it has also *codelivered mm* then following must hold ,

$$VTP(pp)(sender(mm)) \geq VTM(mm)(sender(mm))$$

Thus we add an invariant given at *Inv 11* in Fig. 35 which state that if the vector time of process *p1* is equal or more than vector time stamp of any sent message *m* then *p1* must have *codelivered* the message *m.* Adding *Inv 11* to the model generates proof obligations associated with other events. Discharging these proof obligations required other invariants given as *Inv 12*,*13* and *14*. Therefore, after three iteration of invariant strengthening we arrive at a set of invariants which is sufficient to discharge all proof obligations relating abstract *causalorder* and *vector clock rules.*

Relationship of abstract total order and sequence number

Replacing abstract variable *totalorder* by sequence number in the operations of *SendControl* and guards of *TOdeliver* event generate proof obligations. The first proof obligation which can not be discharged automatically requires us to prove following for *TOdeliver* event.

$$
\begin{array}{l}
TOdeliver(pp, mc)PO2 \\
\left[\begin{array}{l}
mc \in dom(sender) \wedge \\
mtype(mc) = Control \wedge \\
(pp \mapsto messcontrol(mc)) \notin tdeliver \wedge \\
\forall m.(seqno(messcontrol^{-1}(m)) < seqno(mc) \Rightarrow (pp \mapsto m) \in tdeliver) \wedge \\
m \mapsto messcontrol(mc) \in totalorder \\
\Rightarrow \\
(pp \mapsto m) \in tdeliver
\end{array}\right]
\end{array}
$$

It may also be noted that this proof obligation appears due to replacement of following guard of *TOdeliver* involving abstract variable *totalorder*,

$$\forall m.((m \mapsto messcontrol(mc) \in totalorder \Rightarrow (pp \mapsto m) \in tdeliver)$$

by the guard involving variable *seqno*,

$$\forall m.((seqno(messcontrol^{-1}(m)) < seqno(mc) \Rightarrow (pp \mapsto m) \in tdeliver)$$

Therefore, in order to discharge this proof obligation we add the invariant *Inv 15* to our model which relates abstract variable *totalorder* with the concrete *seqno*. This invariant state that if corresponding computation messages of control messages *m1* and *m2* are in *totalorder* then sequence number of *m1* is less then sequence number of *m2*. We notice that this invariant is sufficient to discharge all proof obligations generated by *SendControl* and *TOdeliver* events.

6.6 Second Refinement : Replacing Sequence Number by Vector Clocks

In the second refinement we outline how the need for generating separate sequence numbers can be correctly be implemented by the vector clock. It can be noticed that the *total order* on the messages in the first refinement is realized with the sequence numbers. The events *SendControl* and *TOdeliver* in the first refinement are further refined for the elimination of the need of the sequence numbers to be generated by the sequencer.

The specifications of the *Broadcast* and *CausalDeliver* events of first refinement remains unaltered as none of these events make use of sequence numbers. In the second refinement, the variables *seqno* and *counter* are replaced by vector clock rules. The specifications of the refined *SendControl* and *TOdeliver* events are given in Fig. 36, 37.

As shown in Fig. 36, the operations assigning sequence number to the control messages is removed in the refined *SendControl* event. We use the fact that the vector time stamp of the control message contains enough information required for *todelivery* of the messages. Also as shown in Fig. 37, the guard of the event *TOdeliver* which contains sequencer numbers in abstract model are replaced by vector rules. This replacement in the refinement generates proof obligations involving *seqno* and vector time stamp of messages.

To prove these proof obligations we add *Inv 18*, shown in Fig. 38 to our refined model. Adding *Inv 18* to the refinement require us to add new invariants *Inv 19,20* to the refinement. A brief description of these invariants is given below.

```
SendControl (pp ∈ PROCESS , mm ∈ MESSAGE, mc ∈ MESSAGE ) ≙
 WHEN
        $pp = sequencer$
    $\wedge$  $mc \notin dom(sender)$
    $\wedge$  $mm \notin ran(messcontrol)$
    $\wedge$  $mtype(mm) = Computation$
    $\wedge$  $(pp \mapsto mm) \in receive$
    $\wedge$  $\forall(m,p)\cdot( p \in PROCESS \wedge m \in MESSAGE \wedge mtype(m)=Computation$
                  $\wedge VTM(m)(p) \leq VTM(mm)(p) \Rightarrow m \in ran(messcontrol))$
 THEN
        $mtype(mc) := Control$
    ||  $messcontrol := messcontrol \cup \{mc \mapsto mm\}$
    ||  LET $nVTP$ BE $nVTP = VTP(pp) \lhd\!\!- \{ pp \mapsto VTP(pp)(pp)+1\}$
        IN  $VTM(mc) := nVTP$ || $VTP(pp) := nVTP$ END
    ||  $sender := sender \cup \{mc \mapsto pp\}$
 END
```

Figure 36: Second Refinement : SendControl

```
TODeliver (pp ∈ PROCESS , mc ∈ MESSAGE) ≙
 WHEN
        $mc \in dom(sender)$
    $\wedge$ $mtype(mc)=Control$
    $\wedge$ $(pp \mapsto mc) \in buffer$
    $\wedge$ $(pp \mapsto messcontrol(mc)) \notin deliver$
    $\wedge$ $\forall m.( m \in MESSAGE \wedge (VTM(m)(sequencer) \leq VTM(mc)(sequencer))$
                  $\Rightarrow (pp \mapsto messcontrol(m)) \in deliver)$
 THEN
        $buffer := buffer - \{pp \mapsto mc\}$
    ||  $deliver := deliver \cup \{pp \mapsto messcontrol(mc)\}$
 END
```

Figure 37: Second Refinement : TODeliver

- The sequence number assigned to a control message is same as sequencer's own logical time at the time of sending this message*(Inv-16)*.
- For the given two control messages *m1* and *m2*, if the vector time stamp of *m1* is less than the vector time stamp of *m2* then the sequence number given to *m1* is also less than sequence number of *m2* *(Inv-17)*.
- For the given two control messages *m1* and *m2*, if the sequence number given to *m1* is less than sequence number of *m2* then the vector time stamp of *m1* is also less than the vector time stamp of *m2* *(Inv-18)*.

After discharging the proof obligations generated due to the addition of these invariants associated with the events *Broadcast*, *SendControl* and *TOdeliver*, we ensure that events in Fig. 36, 37 are valid refinement of events in Fig. 34.

The overall proof statistics for the development of a model of Total Causal Order is given in Table 4.

	Invariants	Required By
/*Inv-16*/	$mtype(m) = Control \wedge (m \mapsto sequencer) \in sender$ $\Rightarrow seqno(m) = VTM(m)(sequencer))$	SC,TOD
/*Inv-17*/	$mtype(m1) = Control \wedge mtype(m2) = Control$ $\wedge\ VTM(m1)(p) \leq VTM(m2)(p)$ $\Rightarrow seqno\ (m1) \leq seqno\ (m2)\ )$	BC,SC,TOD
/*Inv-18*/	$mtype(m1) = Control \wedge mtype(m2) = Control$ $\wedge\ seqno\ (m1) \leq seqno\ (m2)$ $\Rightarrow\ VTM(m1)(p) \leq VTM(m2)(p))$	SC,TOD

Figure 38: Second Refinement : Gluing Invariant

Machine	Total POs	Completely Automatic	Required Interaction
Abstract Model	92	56	36
Refinement-I	51	32	19
Refinement-II	20	08	12
Overall	163	96	67

Table 4: Proof Statistics-Total Causal Order

7 Conclusions

The group communication primitives has been studied as a basic building block for the development of fault tolerant distributed services. These primitives provide higher guarantees on the delivery of the messages. In this paper we have outlined the formal development of a distributed system in Event-B which provides a total and a causal order on the messages. Firstly we have developed a model of a causal order broadcast through incremental development. In this development we have outlined how an abstract causal order on the message is constructed by the sender process. In the refinements we have shown how an abstract causal order can correctly be implemented by a system of vector clock. Subsequently, in the second development, we develop a system of total order broadcast. In this development we outline how an abstract total order is constructed by the sequencer and in the refinement steps we present how the recipient processes delivers messages in the total order. Lastly, we present development of system of total causal order broadcast which satisfies both total and a causal order on the message delivery. In the refinements we outline how the abstract *total order* and *causal order* can correctly be implemented by a vector clock system. In the further refinement we also outline how the requirement of generation of sequence numbers may be eliminated by employing vector clock rule. In the development of these system we also discover interesting invariants which provide a clear insight to the system.

The work reported in [13, 27] applies formal method to the group communication system in order to verify the properties of algorithm. In [13] I/O automata is used for formal modelling and a series of invariants relating state variables and reachable states are proved by hand using the method of induction. Similarly, in [27] the formal results for total and causal order are given which provide a proof of correctness by theorem proving. Instead of theorem proving or proving correctness of trace behavior, our approach is based on defining properties in abstract model and proving that our model of algorithm is a correct refinement of abstract model.

The formal approach considered in this paper is based on Event-B which facilitates incremental development of systems. We have used Click'n'Prove B tool for proof management. This tool generate the proof obligations due to refinement and consistency checking and help discharge proof obligation by the use of automatic and interactive prover. The majority of proofs are discharged by automatic prover however some of the complex proofs require the use of the interactive prover. In the development of a causal order broadcast 67 percent of the proofs are discharged by the prover automatically. Similarly, in the development of total order broadcast and total causal order broadcast respectively 74 percent and 58 percent proofs are discharged by the automatic prover. The proofs helps us to understand the complexity of problem and the correctness of the solutions. They also helps us to discover new system invariants providing a clear insight to the system.

References

[1] J.-R. Abrial. *The B-Book: Assigning programs to meanings.* Cambridge University Press, 1996.

[2] J.-R. Abrial. Extending B without changing it (for developing distributed systems). In H. Habrias, editor, *First B Conference*, November 1996.

[3] Jean-Raymond Abrial. Event based sequential program development: Application to constructing a pointer program. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 51–74. Springer, 2003.

[4] Jean-Raymond Abrial and Dominique Cansell. Click'n' Prove: Interactive proofs within set theory. In *TPHOLs*, pages 1–24, 2003.

[5] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A mechanically proved and incremental development of ieee 1394 tree identify protocol. *Formal Asp. Comput.*, 14(3):215–227, 2003.

[6] Roberto Baldoni, Stefano Cimmino, and Carlo Marchetti. Total order communications: A practical analysis. In Mario Dal Cin, Mohamed Kaâniche, and András Pataricza, editors, *EDCC*, volume 3463 of *Lecture Notes in Computer Science*, pages 38–54. Springer, 2005.

[7] Roberto Baldoni and Michael Raynal. Fundamentals of distributed computing: A practical tour of vector clock systems. *IEEE Distributed Systems Online*, 3(2), 2002.

[8] K P Birman and R Van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[9] Kenneth P. Birman, André Schiper, and Pat Stephenson. Lightweigt causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, 1991.

[10] Michael Butler. An approach to the design of distributed systems with B AMN. In Jonathan P. Bowen, Michael G. Hinchey, and David Till, editors, *ZUM*, volume 1212 of *Lecture Notes in Computer Science*, pages 223–241. Springer, 1997.

[11] Michael Butler and Maria Walden. Distributed system development in B. Proc Ist Conf. on B Method,Nantes,pp 155-168, 1996.

[12] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[13] Alan Fekete, Nancy A. Lynch, and Alexander A. Shvartsman. Specifying and using a partitionable group communication service. *ACM Trans. Comput. Syst.*, 19(2):171–216, 2001.

[14] V. Hadzilacos and S.Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR 94 -1425, Cornell University,NY, 1994.

[15] M. Frans Kaashoek and Andrew S. Tanenbaum. An evaluation of the amoeba group communication system. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS96)*, pages 436–448, IEEE Computer Society,1996.

[16] Bettina Kemme and Gustavo Alonso. A suite of database replication protocols based on group communication primitives. In *Proc. Intl. Conf. Distributed Computing System, Amsterdam, ICDCS*, pages 156–163, 1998.

[17] Bettina Kemme, Fernando Pedone, Gustavo Alonso, and André Schiper. Processing transactions over optimistic atomic broadcast protocols. In *IEEE Computer Society, ICDCS*, pages 424–431, 1999.

[18] Bettina Kemme, Fernando Pedone, Gustavo Alonso, André Schiper, and Matthias Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE Trans. Knowl. Data Eng.*, 15(4):1018–1032, 2003.

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

[20] Leslie Lamport and Nancy A. Lynch. Distributed computing: Models and methods. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 1157–1199. 1990.

[21] Michael Leuschel and Michael J. Butler. Pro B : A model checker for B. In *FME*, pages 855–874, 2003.

[22] C Metayer, J R Abrial, and L Voison. Event-B language. RODIN deliverables 3.2, http://rodin.cs.ncl.ac.uk/deliverables/D7.pdf, 2005.

[23] Michel Raynal, André Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters.*, 39(6):343–350, 1991.

[24] Michel Raynal and Mukesh Singhal. Logical time: Capturing causality in distributed systems. *IEEE Computer*, 29(2):49–56, 1996.

[25] Abdolbaghi Rezazadeh and Michael J. Butler. Some guidelines for formal development of web-based applications in b-method. In Helen Treharne, Steve King, Martin C. Henson, and Steve A. Schneider, editors, *ZB*, volume 3455 of *Lecture Notes in Computer Science*, pages 472–492. Springer, 2005.

[26] Carlo Marchetti Stefano Cimmino and Roberto Baldoni. A classification of total order specifications and its application to fixed sequencer-based implementations. *Journal of Parallel and Distributed Computing*, 66(1):108–127, January 2006.

[27] C. Toinard, Gerard Florin, and C. Carrez. A formal method to prove ordering properties of multicast systems. *ACM Operating Systems Review*, 33(4):75–89, 1999.

[28] Divakar Yadav and Michael Butler. Rigorous design of fault-tolerant transactions for replicated database systems using event b. In Michael Butler, Cliff B. Jones, Alexander Romanovsky, and Elena Troubitsyna, editors, *RODIN Book*, volume 4157 of *Lecture Notes in Computer Science*, pages 343–363. Springer, 2006.