THE PUBLISHING HOUSE
OF THE ROMANIAN ACADEMY

PROCEEDINGS OF THE ROMANIAN ACADEMY, Series A,
Volume 13, Number 3/2012, pp. 261–268

SPECIFYING AND MODELING MULTICAST COMMUNICATION IN CBCAST PROTOCOL

Seyed Morteza BABAMIR

University of Kashan, Department of Computer Engineering, Kashan, Iran
E-mail: babamir@kashanu.ac.ir

Reliability of the message passing in *multicast communications* depends on ordering of messages in conformance with properties of used communication protocol. Among others, CBCAST is the protocol used for ordering and synchronization of messages in multicast communication. This study aims to discuss verification of message passing in the multicast communication against properties of the CBCAST protocol. To this end, The CBCAST properties are expressed using Temporal Logic and then they are mapped to state machines. When a message is multicasted to some group of networked clients, the message states are followed and checked against the state machine constructed for a property of the CBCAST protocol.

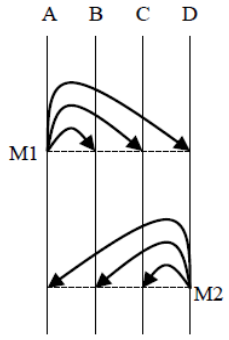
Key words: multicast communication, message ordering, CBCAST protocol, temporal Logic.

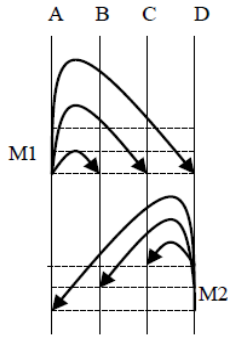
1. INTRODUCTION

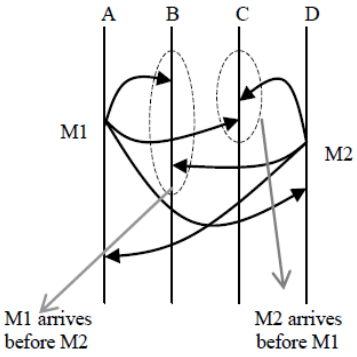
Multicast communication is a one-to-many message passing where a node sends a message to a *group* of nodes. When a message is sent to a group, group members should be received the message in a consistent order. Generally, groups are formed based on network IP addresses; so, for each group a special network address is considered to which multiple machines (group members) listen. When a message is sent to such an address, it is automatically delivered to all machines are listening in the address. Accordingly, in a network there are different groups that each group has a different multicast address.

However, consistency in message delivery to group members is a concern. The consistency means the time order in which messages should be delivered. The consistency may be respected in three types of *synchronizations*: (1) Hard, (2) Soft and (3) Weak. The hard synchronization denotes that delivery time of a message to group members is exactly same as broadcast time of the message. Due to the network latency, such an ordering is not easy to implement (Fig.1). The soft synchronization denotes that if broadcast time of a message is close to the broadcast time of another one, it is not important that which message as the first one is delivered to group members. Just it should be guaranteed that messages are delivered to each group member in same order; however, the order may not be the sent order (Fig 2). The weak synchronization denotes a relaxation of ordering in delivery of messages to group members (Fig. 3) [1]. This synchronization is employed by the *CBCAST* protocol [2]. Since many message passing systems use the weak synchronization for the sake of performance, this paper discuss just verification of message ordering against the weak consistency.

If we consider broadcasting and delivering messages as events, we can exploit an event based formalism to specify message ordering. To this end, we use *Temporal Logic* [3]. The logic appeared in: (1) time instant and (2) time interval. In this paper, interval temporal logics: (1) Future Interval Logic (*FIL*) and (2) Graphical Interval Logic (*GIL*) [4, 5] are used, where they have *textual* and *graphical predicates* respectively. Having specified message ordering by the temporal logics, we construct a verifier automaton to verify weak synchronization of messages delivered to group members. The automaton is created using *Statechart* [6] in UML featuring *hierarchical* and *concurrent* states (Fig. 4).







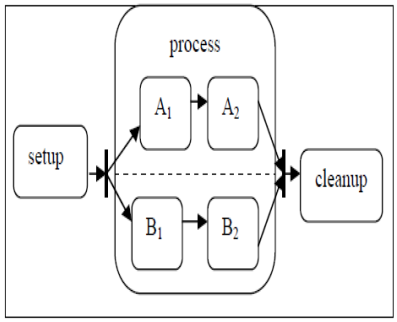


Fig. 1 – Hard synchronization. Fig. 2 – Soft synchronization. Fig. 3 – Weak synchronization. Fig. 4 – Samples Statecharts.

In Fig. 4, states A1, A2, B1 and B2 are nested in state process. States A1 and A2 and also states B1 and B2 are sequential while state A1 or state A2 is concurrent with state B1 or state B2. Similarly, state B1 or state B2 is concurrent with state A1 or state A2. The dotted line in Fig. 4 shows two concurrent areas consisting of states A1 and A2 and states B1 and B2. Having exited from state setup in Fig. 4, system concurrently holds in states A1 and B1. Similarly, the system just makes a transition to state cleanup when it exits both state A2 and state B2. In this paper, Statecharts are used to state constraints should govern message ordering and synchronization.

Since each group member in multicast communication may be on a distributed machine, the delivery of messages may fail because of network links disconnections or network hosts failures. Moreover, routing delays may cause to violation of message ordering among recipient members. This means that two recipient members of a group may receive two messages, say m1 and m2, in a different ordering.

Contribution of this study in verifying multicast message communication is proposing a constructive approach including two connected methods: (1) formal definition of governing constraints in the multicast communication and (2) extraction of verifier automata from the formal definitions. The constructed automata are used to dynamic verification of delivered messages to group members against the governing constraints. This is significant because static verification of message passing between communicating processes faces a large and complex set of possible states while dynamic verification claims verification of just current situation. This leads to creation of the downscale automata are more clear and cheap to verify.

The paper is continued as follows. Section 2 discusses related work and Section 3 explains CBCAST protocol. Temporal logics and their application in demonstrating message orderings are discussed in Section 4. Automata are dealt with in Section 5 and in Section 6 the suggested approach for verification of message communications are put forward. In this approach, properties of CBCAST protocol are specified using temporal logics and then automat are created from the logics. The ordering and synchronization of broadcasted messages are verified using automata. Finally, in Section 6 conclusions are discussed.

2. MULTICAST COMMUNICATION

Multicasting group communication is a message passing system in which a source sends messages to members a group. The members may be distributed among machines of a network known by some IP address. A *hard* synchronous multicast system is one in which messages are delivered to the group members sequentially and with zero time to complete [7]. Fig. 1 shows a hard synchronization where message M_1 was delivered simultaneously to members *B*, *C* and *D*. However, enforcement of hard synchronization is hard to implement. This justifies more light synchronization mechanisms because in many multicast message passing systems, absolute time ordering is not need. Fig. 2 shows *soft* synchronization where messages M_1 and M_2 are delivered close together in time. Note that it should be assured that messages are delivered to all group members in *same order*. In soft synchronization, message delivery takes a small amount of time; however, all messages should be delivered to the members in the same order. Although implementation of such synchronization is feasible, some multicast message passing systems can accept weaker synchronization to enjoy more performance.

Fig 3 shows weak (virtual) synchronization [7] where the orderings messages delivered to processes *B* and *C* are not same; for process B, ordering of delivery is M_1M_2 while for process *C* is M_2M_1. Such reverse ordering is acceptable if messages M_1 and M_2 are *concurrent* not *causally related* [7]. Two messages are concurrent if they are unrelated, i.e. none of them depends on the other. Two messages are causally related if the second message caused by the first one. This means that two messages M_1 and M_2 are causally related if process *B* sends message M_2 to process *C* when process B received message M_1. Therefore, weak synchronization means that causally related messages should be delivered in the same order. However, the system is free to deliver concurrent messages in a different order to different group members. The weak synchronization is considered by the CBCAST protocol where a group consists of *n* processes. Each group member (process), say *i* (n-1 ≥ *i* ≥ 0), has a *clock vector* consisting of *n* clocks where each clock denotes ordering of the message sent by process *i*.

Fig. 5 shows three processes P_0, P_1 and P_2 including clock vectors VC_0, VC_1 and VC_2 respectively. Each vector consists of three clocks for processes P_0, P_1 and P_2. When process P_0 intends to send message *m* to processes P_1 and P_2, it increases its clock in VC_0 by 1 (so, vector clock VC_0 is changed from (0,0,0) to (1,0,0)) and sends VC_0 with message *m* to process P_1 and P_2. Having received message *m* and vector clock VC_0, process P_1 intends to send message m^* to processes P_0 and P_1. To this end, it set VC_1 to VC_0 and increases its clock in VC_1 by 1 (so, vector clock VC_1 is changed from (0,0,0) to (1,1,0)) and sends m^* with VC_1 to processes P_1 and P_2. As this scenario shows, messages *m* and m^* are causally related; so, they should be delivered to process P_2 in order. As Fig. 5 shows, since due to network delays, the second message, m^*, is reached before the first message, *m*, the system middleware should delay delivery of message m^* to process P_2 until after delivery of message *m*.

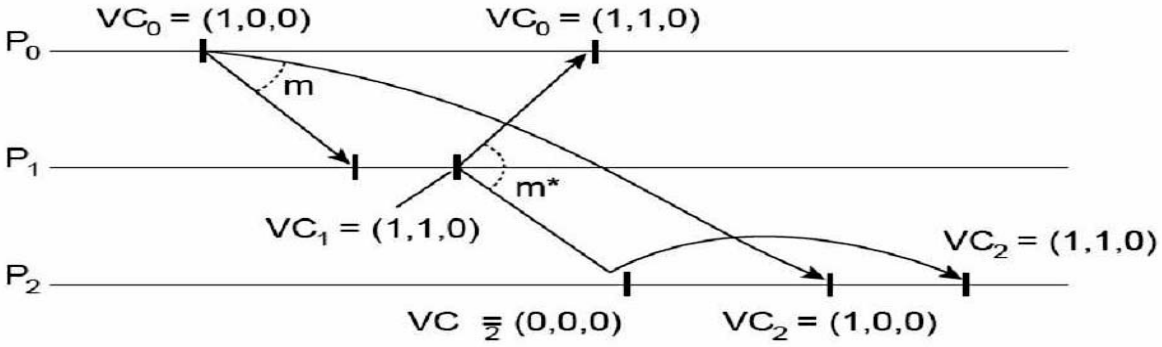


Fig. 5 – Enforcement of causal ordering of messages using vector clocks.

3. TEMPORAL LOGIC

Temporal logic formalism [3] is used to show ordering of events using operators, ○, □, ◊ and *U* where denote *next time*, *sometimes*, *always* and *until* respectively. The Propositional Temporal Logic (PTL) is shown as Relation 1 [7] where *p* is proposition and *f* and *f* ' and *f* " are formulas. Propositions ○*f* ', □*f* ' and ◊*f*' mean that *f* ' holds *at the next time*, *always* and *sometimes*. Proposition *f*'*U f* " means that *f* ' holds until *f* " holds.

$$f ::= p \mid \neg f' \mid f' \wedge f'' \mid \ldots \qquad f ::= \circ f' \mid \Box f' \mid \Diamond f' \mid f' U f''. \tag{1}$$

To reason in periods of times, Interval temporal logic (ITL) is introduced. Future Interval Logic (FIL) [4] and Graphical Interval Logic (GIL) [5] are two types of *ITL*. In FIL, a formula is presented by *If*, indicating interval and formula respectively and a closed interval is presented by [θ1|θ2] where θ1 and θ2 are endpoints.

Graphical Interval Logic (GIL) is a visual formalism to show FIL where the formulas are presented graphically. It enables the user to visualize the relative temporal ordering of states by a horizontal dimension indicating the passage of time. Fig. 6 show types of intervals where *f*, *g* and *h* are events and *i* and *p* are

predicate and formula respectively. The first interval states that formula *p always* holds and the second interval states that formula *p sometime* holds. The intervals are in form of *closed-open*, which *close* means that formula *p* always/sometime holds from the beginning of the interval onwards and *open* means that formula *p* always/sometime holds before the end of the interval. The third interval states that predicate *i* holds when event *f* has occurred but it holds no longer when event *h* occurs after occurring event *g*. Similarly Fig. 6d shows a GIL diagram where formula *p* holds in the interval denoted by two consecutive events *q* [5]. Intervals may be nested to express complex temporal relationships.

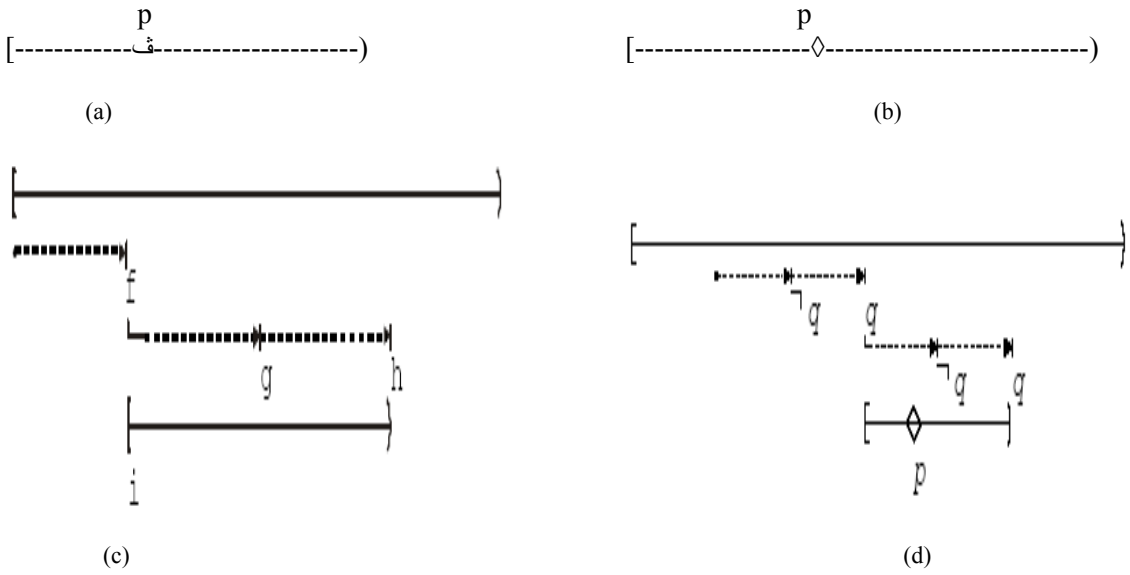


Fig. 6 – Sample GIL diagrams.

4. AUTOMATA

Automata, used as means for capturing complex dynamic behavior, may be represented in different structures. One of them is called *Statecharts* used in the UML modeling language [6, 8]. Statecharts are an extension of conventional finite state machines where states may be nested and concurrent. UML Statecharts consist of three elements: (1) states, (2) transitions and (3) actions. States are elements that persist for a period of time. A transition is represented as *event[condition]/action* where *event [condition]* is the trigger part. The *action* may generate new events and change system variables. By transitions the system is able to change its states in response to environment events. Actions may be simple statements or invoking operations. To specify timing aspects of system behavior, *Timeout* events and *scheduling* actions are used.

Fig. 7 shows a sample Statechart explaining a course to be taken by a student. Three tasks should be achieved *concurrently* by the student for passing the class: Laboratory, Project and Final Test. The dotted lines denote concurrency. Superstate *Taking Class* includes states, *Incomplete*, *Passed* and *Failed* and superstate *Incomplete* includes three concurrent states (*Lab1*, *Lab2*), *Term Project* and *Final Test*. By entering superstate *Incomplete*, a student holds in states *Lab1*, *Term Project* and *Final Test* concurrently.

5. THE PROPOSED METHOD

The proposed method considers weak synchronization and dealt with verification of multicast messages against constraints of weak synchronization. Weak synchronization guarantees reliable message passing when causally related constraint is respected. The proposed method consists of two stages: (1) specification of weak synchronization in Interval Temporal Logic and (2) construction of verification automata from the logical specifications. Now, two constraints of weak synchronization are stated. (1) Each

message is sent by some process, it is eventually delivered in an interval of time. Fig. 8 shows specification of this constraint in FIL (Relation 2) and GIL and Fig. 9 shows the verifier automaton. (2) Enforcement of causal ordering where causally related message m_2 should be delivered before message m_1. Fig. 10 shows this constraint in FIL (Relation 3) and GIL and Fig. 11 shows the verifier automaton.

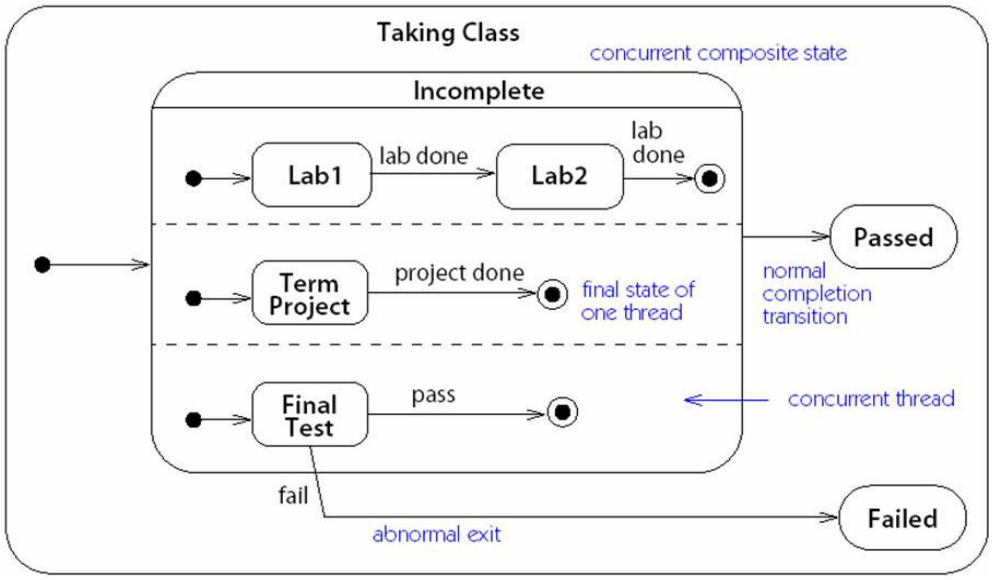


Fig. 7 – A sample Statechart including nested and concurrent states.

As shown in Figs. 8 and 10, a time interval starts when a message is sent and it ends when delivery deadline of the message runs out. The first interval indicated by "[---]" speaks of time passage. Sequence "~$send_m$…$send_m$" and "exp" speak of event "sending a message" and event "expiration". The second interval indicated by "[==)" denotes "wait time" for receiving acknowledgement. It states that acknowledgement should be received before expiration of the considered deadline. In fact, the receiver process should eventually acknowledge before running out the interval.

Now, we explain the verifier automata (Fig. 9) for the first constraint (Fig. 8) where behavior of sender and receiver processes to be verified concurrently. The verifier automata presented in form of Statecharts have two concurrent behavior where the dotted line shows concurrency line. Processes sender and receiver hold in *idle* state when multicast message starts. Notation *do* in Statechart indicates condition of staying in the state; so, notation "do[~send]" in the sender side indicates that the sender in state *idle* while it does not send some message. When event "smsg(m)" occurs, the sender leaves state *idle* and enters state *sent*. In state *sent*, "do[income]" indicates that the sender stays in state *sent* while it is busy with sending the message. When sending message is completed (denoted by event "smsg(m)[comp]", the sender leaves state *sent* and enters state *wait*. The sender stays in state *wait* while it receives no acknowledgment. By expiring wait deadline, sending message *m* fails otherwise there is a normal termination (denoted by two nested bubbles). In the verifier automatoa, send and receive states are considered concurrently.

Similar to sender process, the receiver process holds in *idle* state initially. Notation "do[~recv]" in the receiver side indicates that the receiver in state *idle* while it does not receive any message. When event "recv(m)" occurs, the receiver leaves state *idle* and enters state *recvd*. In state *recvd*, "do[income]" indicates that the receiver stays in state *recvd* while it is busy with receiving message *m*. When receiving message *m* is completed (denoted by event "rcv(m)[comp]", the receiver leaves state *recvd* and there is a normal termination.

Now, we explain specification of the causal ordering constraint stated in GIL and FIL (Fig. 10) and its connected verifier automata (Fig. 11). Fig. 10 shows the interval initiated by events, sending message m_1 and sending message m_2 indicated by "send(p,q,m_1), send(p,q,m_2)" and ended by events receiving acknowledgement ack_1 and receiving acknowledgement ack_2. In Fig. 10, notation "◊" with ack_1 or ack_2 in the

intervals states that acknowledgement ack_1 or ack_2 should be received eventually. Explanation for the verifier automata in Fig. 11 is similar to the verifier automata in Fig. 9. The difference is that receiving causally related messages m_1 and m_2 fails if message m_2 is delivered before message m_1.

Fig. 8 – Specification of the first constraint in Interval Temporal Logics.

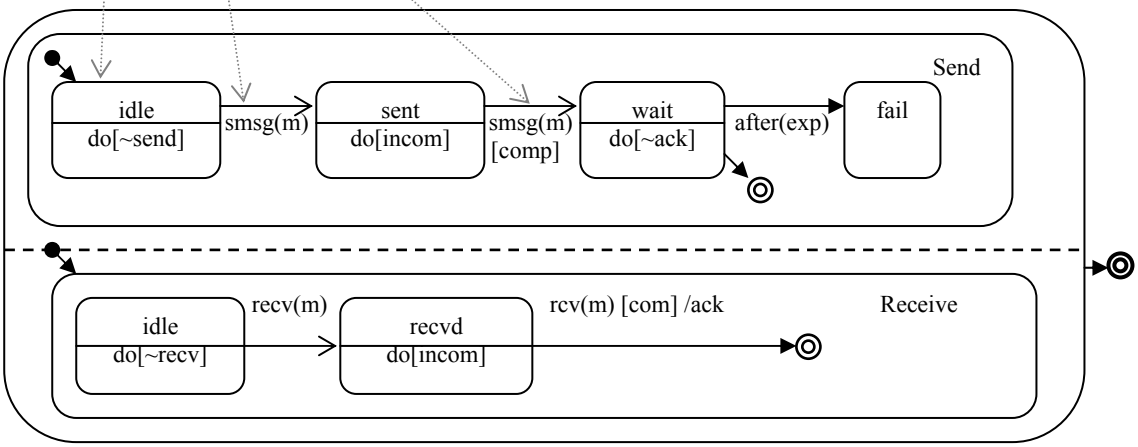


Fig. 9 – Verifier automaton the first constraint.

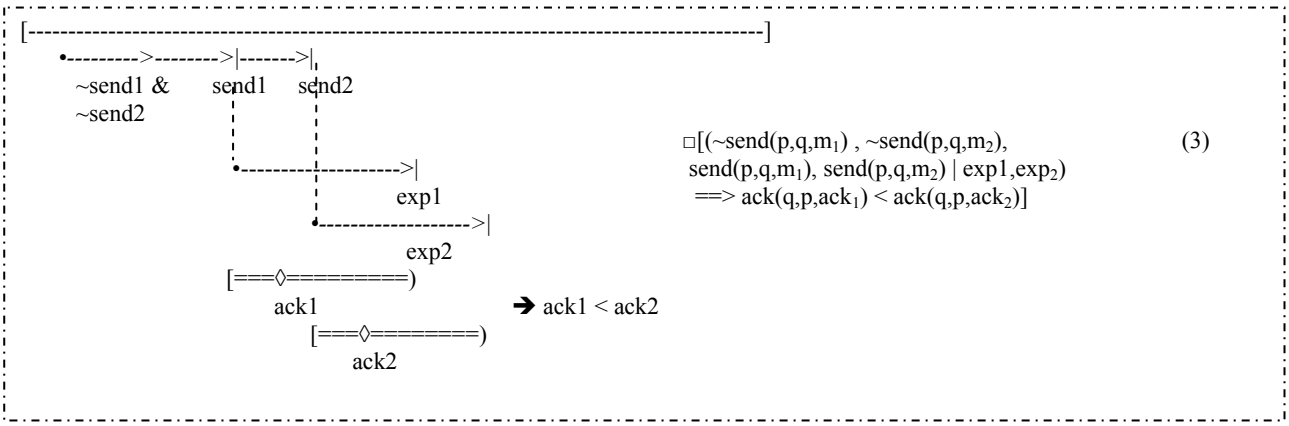


Fig. 10 – Specification of causal ordering in Interval Temporal Logics.

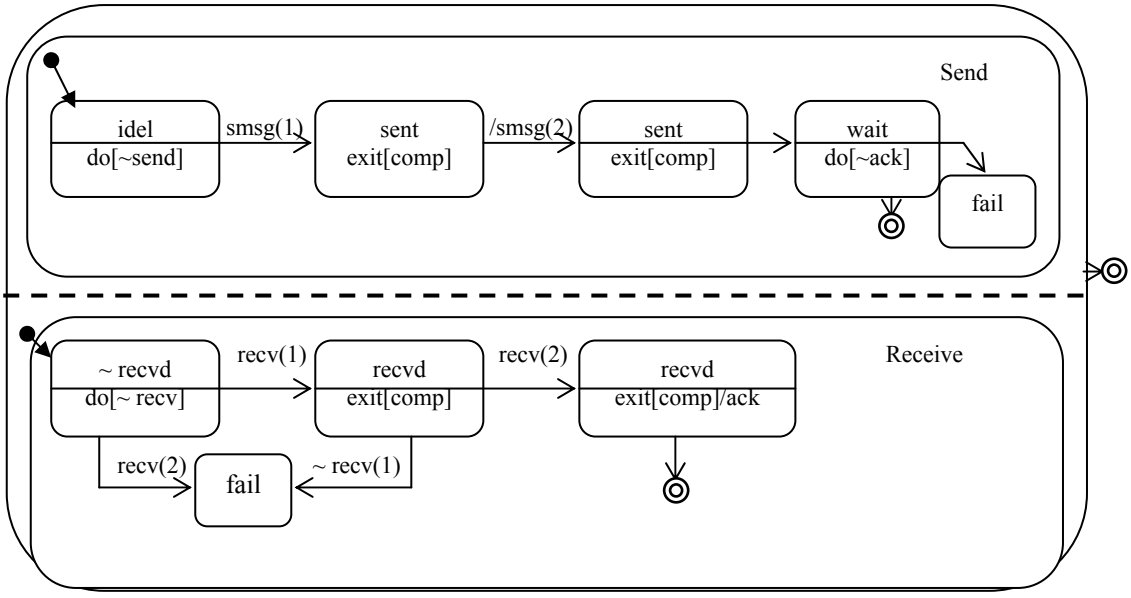


Fig. 11 – Verifier automaton for causal ordering.

6. RELATED WORK AND CONCLUSIONS

This section aims to discuss the related work and conclusions including contribution of this study. message communications are used in: (1) High-level message-oriented applications such as group communications and message-oriented middleware (MOM)[9], (2) object communication technologies such as Remote Method Invocation (RMI) [10], Distributed Component Object Model (DCOM) [3], and Common Object Request Broker Architecture (CORBA) [11] and (3) service-oriented applications such as Service Oriented Architecture (SOA) [12]. However, message communication is decidedly subject to failures such as crashes of message sender or message receiver programs, network disconnection and losing messages. Such cases lead to failure or delay in delivery of messages. In [7], Tanenbaum shows how these failures may lead to false deadlock in resources allocation.

Some studies dealt with the implementation view of multicast message passing and put forward some analyses for java codes [13]. Jonson presented a method for specification and verification of constraints safety and liveness in message communications [14]. He used Labeled Transition System (LTS) to specify constraints of message synchronization where sequences of multicasting messages are verified by a finite automaton as acceptors of finite strings. He tried to use the verification method reducing the verification process. Pilotto and White presented a method that uses PVS formal method for verification of a class of systems where agent communication is accomplished through message passing mechanism [15]. They discussed challenges of formal message passing verification using linear programming. In [16], we presented an approach for verification of multicasting messages using Petri-Nets. In [17–19] Yadav et al presented an approach to specification of causal message ordering using formalism Event-B.

The proposed method in this paper used interval logics to specify constraints of CBCAST protocol. Afterwards, runtime verifier automata were constructed from the logical specifications. While messages are being sent and received, they can be verified at runtime by the automata. The fail states showed that messages are not delivered correctly. While *static* verification of multicasting messages proposed above by others meets verification of a large and complex set of states, the *runtime* verification proposed in this paper dealt with verification of *current* message passing. This scales down verification of states of sender and receiver when just they are busy with sending and receiving messages.

ACKNOWLEDGMENTS

The author is grateful to University of Kashan for supporting this work by Grant No (16124/5).

REFERENCES

1. L. E. MOSER, Y. AMIR, P. M. MELLIAR-SMITH, D. A. AGARWAL, *A History of the Virtual Synchrony Replication Model, Replication: Theory and Practice*, Lecture Notes in Computer Science, in Charron-Bost, Bernadette, Pedone, Fernando, Schiper and Andre (Eds.), Springer, **5959**, pp. 91–120, 2010.
2. K. P. BIRMAN, *The Virtual Synchrony Execution Model*, Reliable Distributed Systems, Technologies, Web Services, and Applications, Springer, 2005.
3. F. KRÖGER AND S. MERZ, *Temporal Logic and State Systems*, Springer, 2008.
4. Y. S. RAMAKRISHNA, P. M. MELLIAR-SMITH, L. E. MOSER, L. K. DILLON AND G. KUTTY, *Interval Logics and Their Decision Procedures*. *PART I: An Interval Logic, Theoretical Computer Science*, **166**, *1–2*, pp. 1–47, 1996.
5. S. RAMAKRISHNA, G. KUTTY, P. M. MELLIAR-SMITH, L. K. DILLON, *A Graphical Environment for Design of Concurrent Real-Time Systems*, ACM Transactions on Software Engineering and Methodology, **6**, *1*, pp. 31–79, 1997.
6. D. DRUSINSKY, *Modeling and Verification Using UML Statecharts, A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking*, McGraw-Hill, 2006.
7. A. S. TANENBAUM, *Distributed Systems: Principles and Paradigms*, Prentice Hall, 2006.
8. *** *OMG Unified Modeling Language Specification*, Version 1.5, March 2003.
9. S.S. ALWAKEEL AND H.M. ALMANSOUR, *Modeling and Performance Evaluation of Message-oriented Middleware with Priority Queuing*, Information Technology Journal, **10**, pp. 61–70, 2011.
10. W. GROSSO, *JAVA RMI*, O'Reilly Media, 2001.
11. *** http://www.corba.org/
12. B.V. KUMAR, P. NARAYAN, T. NG, *Implementing SOA Using Java EE*, Addison-Wesley, 2010.
13. M. ALEKSY, A. KORTHAUS, M. SCHADER, *Implementing Distributed Systems with Java and CORBA*, Springer, 2005.
14. B. JONSSON, *Compositional specification and verification of distributed systems*, ACM Transactions on Programming Languages and Systems, **16**, *2*, pp. 259–303, 1994.
15. C. PILOTTO AND J. WHITE, *Verification of Faulty Message Passing Systems with Continuous State Space in PVS*, Second NASA Formal Methods Symposium, USA, 2010.
16. S. M. BABAMIR, *Constructing Formal Rules to Verify Message Communication in Distributed Systems*, The Journal of Supercomputing, **59**, *3*, pp. 1396–1418, 2011.
17. D. YADAV AND M. BUTLER, *Application of Event B to Global Causal Ordering for Fault Tolerant Transactions*, Workshop on Rigorous Engineering of Fault Tolerant Systems (REFT2005), pp. 93–102, 2005.
18. D. YADAV AND M. BUTLER. *Formal Specifications and Verification of Message Ordering Properties in a Broadcasting System using Event B*, Technical Report, School of Electronics and Computer Science, University of Southampton, 2007.
19. N. FULMARE AND D. YADAV, *Rigorous analysis of byzantine causal order using Event-B*, Proceedings of the International Conference and Workshop on Emerging Trends in Technology, 2010, pp. 723–726.

Received July 26, 2011