Formal Specification and Analysis of Termination Detection by Weight-throwing Protocol

Imran Riaz Hasrat, Muhammad Atif
Department of Computer Science
and Information Technology
The University of Lahore
Lahore, Pakistan

Muhammad Naeem
Department of Electronics
and Electrical Systems
The University of Lahore
Lahore, Pakistan

***Abstract*—Termination detection is a critical problem in distributed systems. A distributed computation is called terminated if all of its processes become idle and there are no in-transit messages in communication channels. A distributed termination detection protocol is used to detect the state of a process at any time, i.e., terminated, idle or active. A termination detection protocol on the basis of weight-throwing scheme is described in Yu-Chee Tseng, "Detecting Termination by Weight-throwing in a Faulty Distributed System", JPDC, 15 February 1995. We apply model checking techniques to verify the protocol and for formal specification and verification the tool-set UPPAAL is used. Our results show that the protocol fails to fulfil some of its functional requirements.**

***Keywords*—Termination detection; weight-throwing protocol; formal specification and verification; model checking**

I. Introduction

Termination detection is an important problem for distributed systems. For a distributed system, termination detection is based on the concept of a process state. During a distributed computation, a process can either be in alive or dead state. An alive state means that a process is still performing its task whereas dead state represents that the process becomes idle simultaneously. Dead and alive states are referred as *passive* and *active* states as shown in Fig. 1. At the start of the computation, all the processes are supposed to be in *active* state. Processes can take several actions discussed below:

- Only the processes in *active* state can send *basic* messages to other processes.
- Any *active* process can reach a *passive* state at any time.
- A *passive* process becomes *active* again by receiving a *basic* message.

A distributed computation is called terminated if all of its processes become *passive* and there are no in-transit messages in the communication channels.

Many applications of distributed systems depend on termination detection of a computation to guarantee a proper operation. In multiphase algorithms [2], one phase depends on proper completion of other phase. So, initiation of new phase needs termination detection of previous phase. In distributed databases, deadlock detection is a critical problem and this problem is purely related to termination detection [3]. Garbage collection [4] and token loss detection in a token ring are other examples of termination detection problems. Termination detection solution allows a system to guarantee that all tasks in the system are obviously complete and this permits the dependent systems to start their computations. About more than three decades ago, termination detection problem was separately suggested by Dijkstra and Scholten [5] and Francez [6]. Many researchers started to tackle this problem by developing different termination detection algorithms described in [7]–[15].

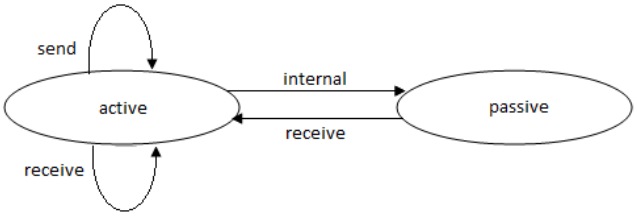

Fig. 1. Termination detection [1].

Formal methods are based on mathematical tools and techniques used for design, specification and verification of different hardware and software systems. Formal methods provide correctness for all requirements and inputs of a given system. In the past, formal methods were used for only safety critical and defense related systems [16]–[18]. Now a days, high demand of error free and secure systems is giving much importance to formal methods.

We present a formal modeling and analysis of the termination detection by weight-throwing in a faulty distributed system presented in [7]. It is a comprehensive analysis of all possible versions of protocol along with verification of detailed functional requirements. Basic concept in weight-throwing protocol is that each process sends some weight with every *basic* message. On reception of this message, recipient process adds this weight to its current weight. We present formal verification of the weight-throwing protocol using a model checker known as UPPAAL.

UPPAAL has a simulator that is used to develop the model [19]. The verifier in UPPAAL has capability to create the traces that can lead the action sequences where system's required

property fails. To investigate this situation, action sequences are replayed in simulator.

We formally model and analyze both parts of the protocol. We present a formal specification of the protocol in the timed-automata language of UPPAAL. Then, we specify functional requirements for safety and liveness of the protocol. We also present specification and verification of its invariants. We also analyze the protocol in the situations when some processes may tend to fail. We discover some situations in which protocol fails to satisfy its functional requirements, termination is not detected in these situations. The protocol also fails to satisfy some of its invariants. We present counterexamples for the requirement violations in the form of message sequence charts.

II. Weight-Throwing Protocol

Weight-throwing protocol works in a distributed system. In this protocol a process sends half of its weight with its message when it communicates to some other process. Let S is the set of processes i.e. $S = \{P_1, P_2, \cdots, P_n\}$. The S is supposed to be fault-free. In fault-free system, a process never fails. A process from S with minimum id is said to be the *leader*. Total weight in the system is 1. Every process is *active* at the start of the computation and weight of each process P_i is $w_i = 1/n$, where n is the cardinality of the set S. The *leader* collects all the weights in the system and announce termination. Upon every message from a process P_i to P_j following actions are performed:

1) P_i divides its weight w_i into two equal positive real parts x, y so that $w_i = x + y$.
2) P_i sends a *basic* message $B(x)$ along with a weight x.
3) P_i updates its current weight as $w_i = y$.
4) On reception of the *basic* message, P_j's weight increases as $w_j = w_j + x$.

Any process in the system can become *passive* at any time. When a process other than the *leader* becomes *passive*, it sends a *control* message to the *leader* for submitting its weight. After sending the *control* message this process sets its weight to 0. Any *passive* process can become *active* again by receiving a *basic* message from some other process. The following are the invariants in the protocol:

- Each process and in-transit messages in the system have a non-zero weight at any time.
- If we take the sum of the weights of all processes and in-transit messages at any time, it is always 1.

When the *leader* becomes *passive*, it accumulates all the weights in the system. If accumulated weight becomes equal to 1, the *leader* calls termination. Weights should be handled precisely. Fractional values of all weights make it nearly impossible to make the sum to 1 again due to rounding errors of float values. This problem can be solved by representing the weight using two integer values as $[1, n]$ instead of $1/n$.

A. Flow-detecting Scheme for Flushing/Freezing of Channels

In case of faulty distributed system, some processes may fail during a computation. Overall weight of all processes can be less than 1 due to holding weights of failed processes and weights carried by undelivered messages in the channels. This problem is solved by introducing a flow detecting scheme. Let H be a subsystem which contains all healthy processes and all their communication channels. During a computation, the weight change in H at time interval I is equal to the difference of weight flowing into H and the weight flowing out of H. With the help of this scheme, the weight information of failed processes can be obtained from the outgoing weight records of healthy processes in the system because each process keeps the record of incoming and outgoing weights.

Assume that the intended system provides the facility of flushing or freezing of channels connected to faulty processes. Flushing or freezing mean preventing and ceasing further communication between a healthy and a faulty process. There is no global clock in the system that makes it very difficult to get global views of the system weight. A *snapshot* is taken to get the global views. The *leader* sends the *snapshot request* to all the healthy processes. On reception of this request, each healthy process flushes or freezes all of its communication channels connected to faulty processes and submits its incoming and outgoing weights to the *leader*. The *leader* uses these incoming and outgoing weight values to calculate the overall weight of the system.

B. Data Structure of Weight-throwing Protocol

Before the formal modeling of the protocol we need to know the specific keywords and data structure used in the protocol. Let P_i be any process in the system where $i = 1 \ldots n$, n is any arbitrary positive number. The data structure for P_i is given in Table I.

III. Modeling in UPPAAL

Our formal specification in UPPAAL has three participants, i.e. the *Termination*, the *MessageBuffer* and the *SnapBuffer*. The main process is the *Termination* process. This process receives and sends messages to other processes to communicate. Each message holds some non-zero weight. A process adds the incoming weight to its current weight when it receives a message. The communication is asynchronous. The *MessageBuffer* process holds the *basic* and *control* messages when receiver is not ready to receive them. These messages are moved to their receivers when they become ready. The *SnapBuffer* process temporarily stores the *snapshot request* messages sent by the current *leader*. It then sends the *snapshot request* messages to all the healthy processes to inform them the faulty set of processes known to the *leader*. This process also temporarily keeps the *snapshot reply* messages sent by the processes to the *leader*. These messages are delivered to the *leader* when it becomes ready.

The protocol has two parts. In first part, the termination detection is done using a fault-free distributed system. In the second part, a faulty distributed system is used to detect proper termination of processes. In faulty distributed system any number of processes may tend to fail. We present the formal specification of both parts separately to check the correctness of the protocol in both cases.

TABLE I. INFORMAL SPECIFICATION OF WEIGHT-THROWING PROTOCOL

$leader_i$	This variable stores id of current *leader*. At the beginning, $leader_i = 1$.
w_i	This field keeps the value of weight for each process P_i. Initially, the value of w_i is $1/n$, where n is the number of processes.
sum_i	This variable records total weight in the system assumed by the *leader*. It is a real number. At the beginning, $sum_i = 1$.
$IN_i[1 \ldots n]$	This is an array of real numbers. The $IN_i[j]$ keeps the record of all weights thrown out from process P_j to P_i. At the beginning, $IN_i[j] = 0$ for all $j = 1 \ldots n$.
$OUT_i[1 \ldots n]$	This is an array of real numbers. The $OUT_i[j]$ stores all the weights thrown to process P_j from P_i. At the beginning, $OUT_i[j] = 0$ for all $j = 1 \ldots n$.
F_i	This array represents a set of faulty processes. If a process P_i knows P_j to be faulty and P_i has flushed or frozen all the channels to P_i then it belongs to F_i . At the start $F_i = \phi$.
SN_i	This array contains set of all processes to which *snapshot request* is to be sent. Let's suppose P_i initiates the *snapshot request*. If a process P_j replies to the request or it is found faulty by P_i then it is removed from the SN_i. Second *snapshot* can be initiated only when SN_i becomes empty. At the start $SN_i = \phi$.
$temp\_sum_i$	This field stores a real number. During the *snapshot*, it temporarily calculates the total remaining weight.
$consistent_i$	This field indicates a boolean value which keeps the record of a *snapshot's* consistency.
$B(x)$	This represents the *basic* message with a weight x.
$C(x)$	This indicates a *control* message. The *control* message is used for reporting the weight x to the *leader* of the system.
$Request(F_i)$	This represents the message for snapshot request that is sent by the current *leader* of the system P_i. With the help of F_i, message receiving process is informed about the set of faulty processes already known to the *leader*.
$Reply(F_i, IN_i, OUT_i)$	This indicates the reply to the *leader's* request message. This reports the state of the replying process.

IV. MODEL1: TERMINATION DETECTION IN A FAULT-FREE DISTRIBUTED SYSTEM

We specify all the processes of fault-free part of the protocol. The *Termination* and the *MessageBuffer* are the participants in Model1. We present the functionality and formal specification of these participants in this section.

A. Channels

This protocol uses four channels which are described below. To model the functionality of termination detection in a fault-free distributed system, we use hand shaking channels. The working of these channels is described below:

1) *basicMessageS:* This channel is very important because the system uses this channel for the *basic* message communications. It sends *basic* messages to the *MessageBuffer* to hold them until their receivers become ready.
2) *basicMessageR:* For moving stored *basic* messages from the *MessageBuffer* to the receiver process, system uses *basicMessageR* channel.
3) *controlMessageS:* This is a channel for *control* message communications. System uses this channel to send *control* messages to the *MessageBuffer* to hold them until the *leader* becomes ready.
4) *controlMessageR:* This channel moves stored *control* messages from *MessageBuffer* to the *leader*.

B. Global Declarations

Some variables and arrays are declared globally so that each participant can access them and use them according to their needs. Table II represents the global declarations and data types for Model1.

TABLE II. GLOBAL DECLARATIONS FOR MODEL1

numberofweights	This is a constant that represents the total parts of weight.
max_weight_limit	This describes the maximum value of weight that can be sent through communication channels.
maxproc	It is a variable that tells the number of concurrent instances of *Termination* process.
max_proc_id	This variable stores the highest id of concurrent instances of the *Termination* process.
leader	This variable keeps id of the current *leader* of the system.
W1 and *W2*	These variables record the weights when a *basic* or a *control* message is received.
Out_arr[] and *In_arr[]*	Two customized data structures *Weight_out* and *Weight_in* are introduced to define the *Out_arr[]* and *In_arr[]* arrays respectively. The *Out_arr[]* keeps the records of out going weights and *In_arr[]* stores the records of incoming weights of each process of the *Termination* process.

C. The Automaton for Termination Process in Model1

The automaton for the *Termination* process is depicted in Fig. 2. The protocol model has a number of parallel processes, each of which is triggered by a certain communication among each other. The specification of the *Termination* process comprises four communicative choices i.e. sending *basic* message, receiving *basic* message, sending *control* message and receiving *control* message.

For overall working of the *Termination* process, we discuss the functionalities which take place between *active* and *A1* states. The *basicMessageS!* sends two weight values of *basic* message to the *MessageBuffer* for any other process. Two weights actually represent the single weight because we are using two values (numerator and denominator) just to avoid floating point errors. The guards for weight values, limit the number of *basic* messages that a process can send to other processes to reduce transition state space. Going from *A1* to *active* state, the *updateOut()* function updates the *Out_arr[]* to record the outgoing weights. Also the *weight[1]* value is doubled because doubling the denominator value, overall weight of a process becomes half. For a process taking a transition from the *active* to *A2* state, the channel *basicMessageR?* receives two weight values and sender id from its *MessageBuffer* to record incoming weight against a specific sender. In next transition, the *updateIn()* function updates the *In_arr[]* to record incoming weight from that specific process. The *updateWeight()* function records the overall current weight of receiver process. Same procedure is followed when going from *passive* to *active* state because both the transitions are identical and perform exactly same functionality. While taking transitions from *active* to *A3* and *A3* to *passive* state, the channel *controlMessageS!* sends weight values of this process alongwith its id to the *leader*. The function *updateIn()*

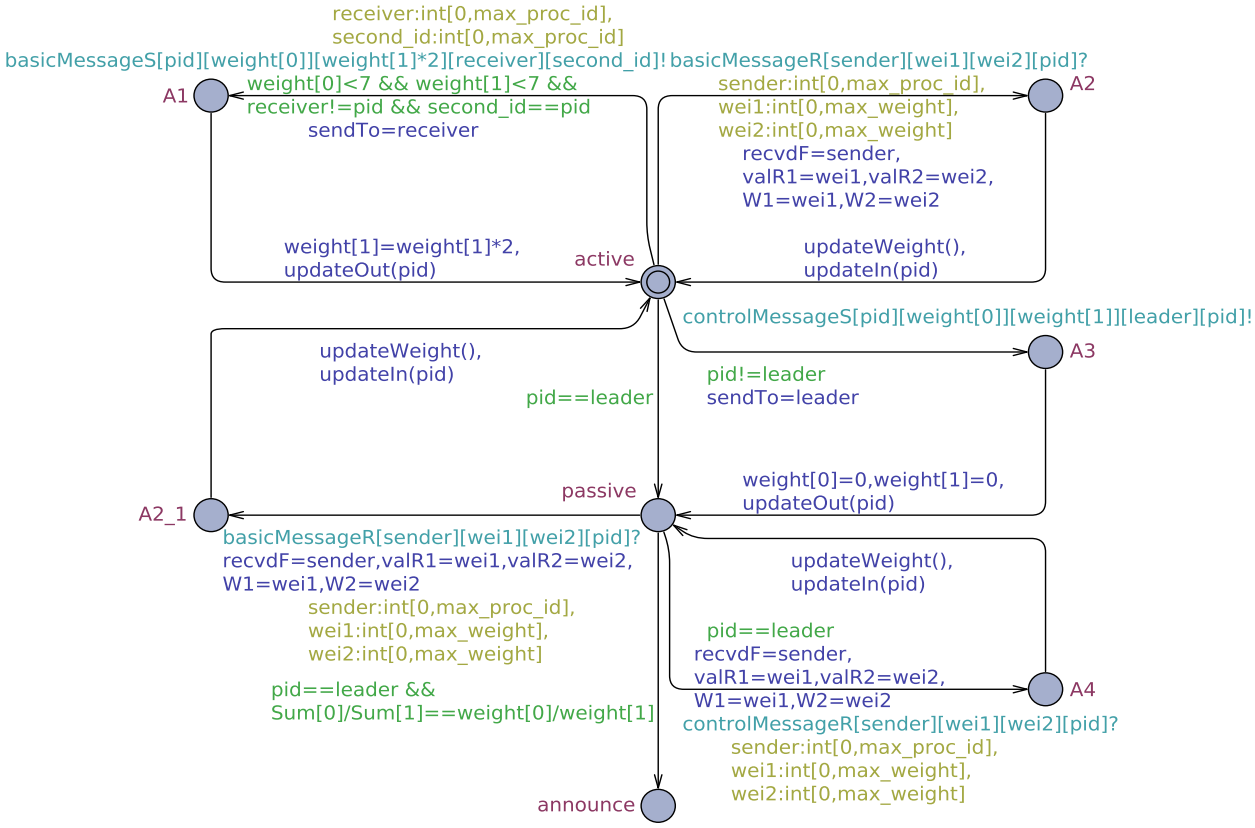

Fig. 2. Formal model for *Termination* process in Model1.

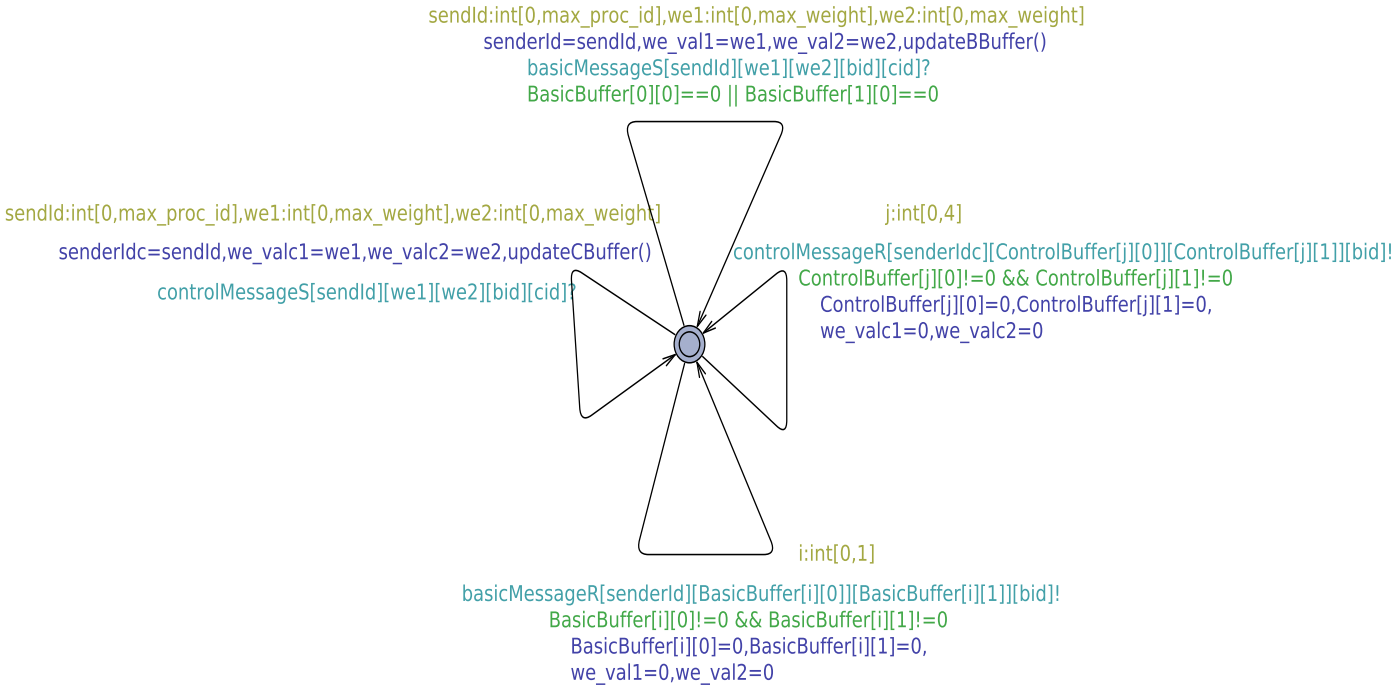

Fig. 3. Formal model for *MessageBuffer* process.

updates *Out_arr[]* of this process. The value of *weight[0]* and *weight[1]* is set to zero to make sure that all the weight is transferred to the *leader*. The working of transitions from *passive* to *A4* is similar to moving from *active* to *A2* state. The only difference is, in *controlMessageR?* channel, the *leader* receives the *control* message from *MessageBuffer*. All other variable definitions are exactly same. Taking transition from *A4* to *passive* state uses exactly same functions as described previously for *A2* to *active* state. Only the *leader* can take the transition from *passive* to *announce* state. The *leader* takes this transition only when it has collected all the weight from all the processes through *control* messages. If this weight becomes equal to the weight defined at the beginning of the computation, the *leader* announces termination.

The local declaration and data types of *Termination* process are given in Table III.

TABLE III. LOCAL DECLARATIONS FOR *Termination* PROCESS IN MODEL1

valR1 and *valR2*	These variables temporarily store first and second value of weight respectively at different transitions.
recvdF and *sendTo*	System uses *recvdF* variable to store process id of the message sender and *sendTo* records process id of the message receiver.
weight[]	*weight* array permanently stores the two weight values for every process at any time.
Sum[]	The *Sum[]* a value container that represents the total weight of the system that is actually 1.
tempIn[]	This array temporarily stores the sum of all incoming weights to a process.
tempOut[]	*tempOut[]* keeps the sum of all the outgoing weights from a process.

In UPPAAL system models, we can declare functions with in the process or alongwith global declarations. The functions can have return types and passing parameters. The *Termination* process also uses some functions to perform its functionality.

1. *updateWeight()*: This function is very important because it is called at different transitions when a *basic* or a *control* message is received. If the receiver has zero weight, the received weight is directly moved to *weight[]* of the process. If the process already has some non-zero weight then function adds the incoming and current weight to calculate the new weight.

2. *updateIn()*: A process uses this function when it receives a *control* or a *basic* message. This function updates the *In_arr[]* to record the incoming weight from a particular process. The incoming weight is directly moved to the *In_arr[]* if receiver is receiving the first message from the sender. The current weight and incoming weight is added to update the incoming weight in *In_arr[]* if the process already has received some messages from the same sender.

3. *updateOut()*: A process calls this function when it sends a *control* or *basic* message. This function updates the *Out_arr[]* to record the outgoing weight to a particular process. The outgoing weight is directly moved to the *Out_arr*array if receiver is receiving the first message from the sender. The current weight and outgoing weight is added to update the *Out_arr[]* if the process already has sent some messages to the same receiver.

D. The Automaton for MessageBuffer Process

The automaton for the *MessageBuffer* process is depicted in Fig. 3. The process has three instances. The specification of the *MessageBuffer* process in the protocol provides four communication choices i.e. receiving a *basic* message from the *Termination* process, sending a stored *basic* message to the *Termination* process, receiving a *control* message from the *Termination* process and sending a stored *control* message to the *Termination* process.

Now we discuss the functionalities for the *MessageBuffer* process when a *basic* message is received through *basicMessageS?* channel. The function *updateBBuffer()* updates the *basic* message buffer to keep the record of this incoming *basic* message until it is not delivered to the respective recipient. The *basicMessageR!* channel sends the stored *basic* message when the receiver *Termination* process becomes ready. The two guards prevent the communication between *MessageBuffer* and *Termination* process when their is no stored message. The function *updateCBuffer()* is called to update the *control* message buffer to keep the record of this incoming *control* message until its recipient is not ready. This happens when a *control* message is received through *controlMessageS?* channel. The *controlMessageR!* sends the stored *control* message when the *leader* becomes ready. The two guards prevent the communication between the *MessageBuffer* and the *leader* when their is no stored *control* message for the leader.

The local declaration and data types of the *MessageBuffer* process are given in Table IV.

TABLE IV. LOCAL DECLARATIONS FOR *MessageBuffer* PROCESS

we_val1 and *we_val2*	The variables store first and second value of *weight[]* for receiving and sending *basic* messages at different transitions of *MessageBuffer* process.
senderId and *senderIdc*	*senderId* variable stores the process id of the *basic* message sender process and *senderIdc* keeps the process id of the *control* message sender process.
we_valc1 and *we_valc2*	These variables contain the first and second value of *weight[]* for receiving and sending *control* messages at different transitions of *MessageBuffer*.
BasicBuffer[] and *ControlBuffer[]*	The *BasicBuffer* array list is to store the two *basic* messages for each process at any time. The limit to store only two *basic* messages is for achieving reduced transition state space. The *ControlBuffer[]* can store five *control* messages for the *leader* process at any time.

The *MessageBuffer* process uses some functions to perform different tasks on different transitions. We discuss these functions below.

1. *updateBBuffer()*: It is called on a transition when a *basic* message is received. It locates the available free space in *BasicBuffer[]* and then stores the incoming *basic* message at that space.

2. *updateCBuffer()*: A transition calls this function when it receives a *control* message at *MessageBuffer* process. The function checks the free space in *ControlBuffer[]* and then stores the incoming *control* message at that position.

V. Model2: Termination Detection in a Faulty Distributed System

We specify all the concurrent processes of faulty part of the protocol. The three participants are the *Termination*, the *MessageBuffer* and the *SnapBuffer*. The functionality and formal specification of these participants is presented in this section. The functionality of the *MessageBuffer* process is already described in Model1. Here we discuss the functionality of remaining participants.

A. Channels

This protocol uses nine channels for communication among processes. Four of them are same as described in Model1. The other five channels are described here in detail.

1) *failReport:* The *Termination* process uses this channel when a process fails. It tells the failed status of a process to other processes. On the other side, a process receives the status of a failed process using this channel.
2) *failRequest:* This channel sends *snapshot request* message to the *SnapBuffer* process.
3) *failRequestR:* The channel sends the stored *snapshot request* message from *SnapBuffer* process to the recipient *Termination* process.
4) *failReply:* This channel is used to send the *snapshot reply* message to the *SnapBuffer* process.
5) *failReplyR:* The channel delivers the stored *snapshot reply* message from *SnapBuffer* process to the recipient *Termination* process.

B. Global Declarations

This system is modeled for termination detection in faulty distributed environment. This is the enhancement of the Model1 (fault-free model). Therefore some global variables are common in both the systems. We present here the description of variables that are not present in Model1 but are present in Model2. The global declarations for Model2 are described in Table V.

TABLE V. Global Declarations for Model2

FIn[] and *FOut[]*	The *FIn* array stores all the incoming weights and *FOut[]* saves the outgoing weights of failed processes which are known to the *leader* during the *snapshot*.
FI_FO_Diff[]	The *FI_FO_Diff* array stores the difference of all the incoming and outgoing weights of failed processes known to the *leader* during the *snapshot*.
S[]	System uses *S* array to store ids of all instances of *Termination* process participating in the system.
F[]	The *F* is a global array. It records the failed processes known to each process. The value may be different for each process.
Ftemp[]	*Ftemp[]* keeps the actual record of failed processes in the system.

C. The Automaton for Termination Process in Model2

We have discussed the *Termination* process for Model1 in previous section. The automaton for *Termination* process in Model2 is depicted in Fig. 4. The specification of *Termination* process has ten communicative choices four of which are same as in Model1. The other six choices are sending *snapshot request* message, receiving *snapshot request* message, sending *snapshot reply* message, receiving *snapshot reply* message, sending *fail report* message and receiving *fail report* message.

Now *Termination* has nine actions from *A1* to *A5* and from *F1* to *F4*. The actions *A1* to *A5* are already discussed in Model1. So we discuss here only the actions *F1* to *F4*. Fig. 5 represents the formal model for *F1*. The *failReport?* channel detects the failed process when no snapshot is in progress. In next transition, the process adds the failed process in its *F[]* and *Flush[]*. The function *Leader()* is called to determine the *leader*. If current process is not the *leader* then it reaches to *active* state. If this process is the *leader* then it reaches to *Snap* state and starts calculating the healthy processes to send them *snapshot request* message. The *SnapBuffer* stores this message until the receiver of this message is not ready. After sending these messages the process is allowed to reach at *active* state.

Fig. 8 represents the formal model for *F2*. A process receives the stored *snapshot request* message from *SnapBuffer* process through *failRequestR?* channel. Then it sends the *snapshot reply* message to the *SnapBuffer* for the *leader* through *failReply!* channel. It records the new *leader*. It matches and updates its *F[]* to record the failed processes known to the *leader*.

Fig. 6 shows the formal model for *F3*. The *leader* receives the stored *snapshot reply* message from *SnapBuffer* process through *failReplyR?* channel. It checks the difference of *F[]* of sender and its own *F[]*. The snapshot is marked as inconsistent if this difference is greater than zero or the snapshot is not consistent. The process which has sent this *snapshot reply* message is removed from *SN[]*. The process reaches to *active* state if a snapshot is in progress otherwise reaches to *Snap* state.

Fig. 7 represents the formal model for *F4*. The *failReport?* channel detects a failed process. The action makes sure that a snapshot is already in progress. In next transition this process adds the failed process in *F[]* and *Flush[]*. It also sets the snapshot as inconsistent. Then after removing the failed process from *SN[]*, if still *SN[]* is non empty then process calls the snapshot.

Fig. 9 represents the formal model for the process when it fails. At failure, the process updates the *FTemp[]* to record its entry in that array and moves from *active* to *fail* state. The channel *failReport!* at *fail* state continuously tells other processes that its status is failed.

We have discussed some local declarations of *Termination* process in Model1. Now we discuss the local declarations of remaining variables in Table VI.

calSN(), *calcDiff()*, *isAvailable()*: The three functions perform the combined functionality of identifying the healthy processes for sending *snapshot request* messages. The function *calSN()* calls the *calcDiff()* function. The *calcDiff()* function calls *isAvailable()* function for every process id. If a true value is returned it means that the process is present in the *F[]* of calling process and there is no need to send *snapshot request* message to that process.

AddIn(): The function *AddIn()* calculates the total incoming weights of all the failed processes known to the current *leader* during the *snapshot*. It adds the incoming weights of every failed process to make a sum of incoming weights in *FIn[]*.

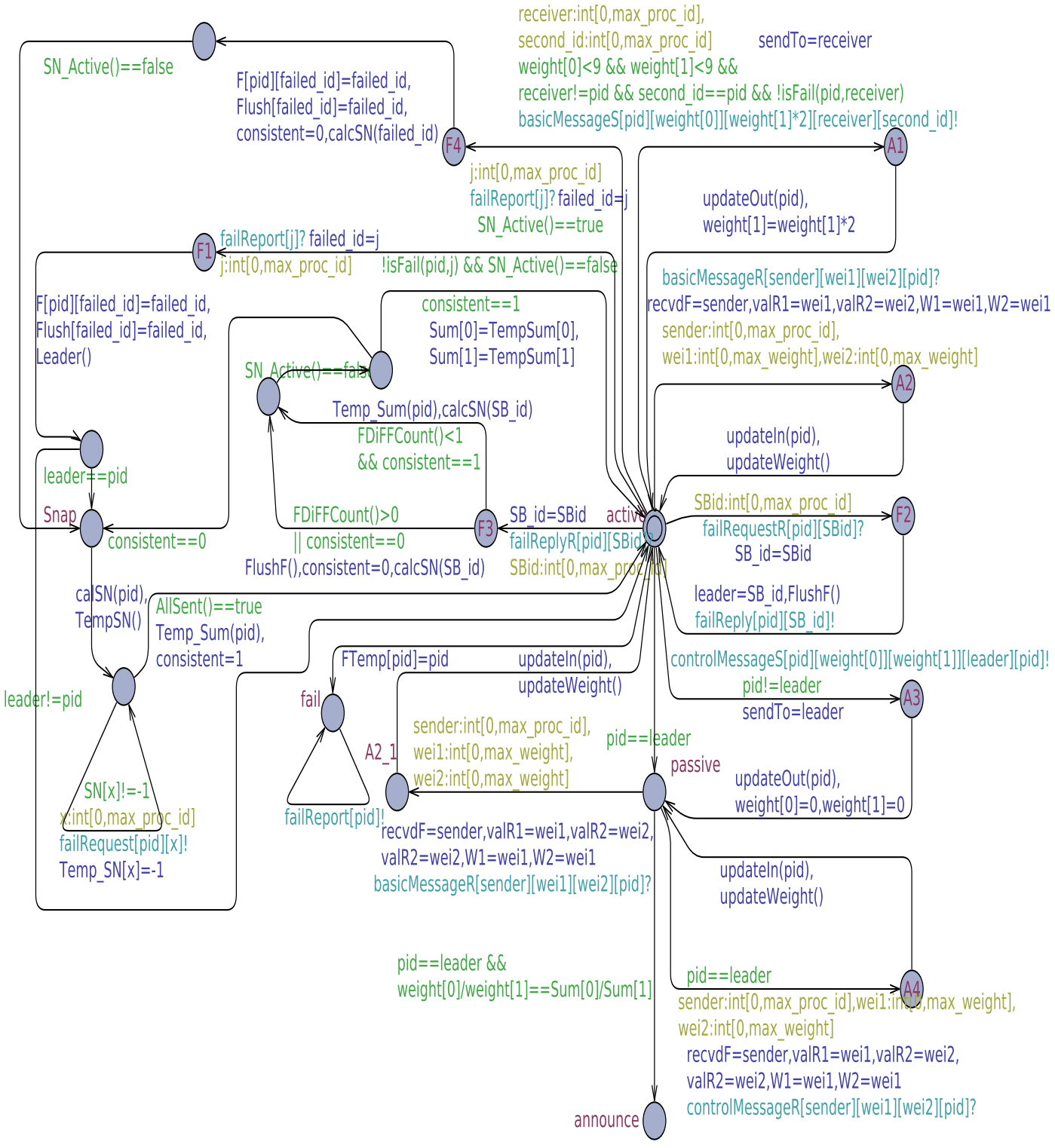

Fig. 4. Formal model for *Termination* process in Model2.

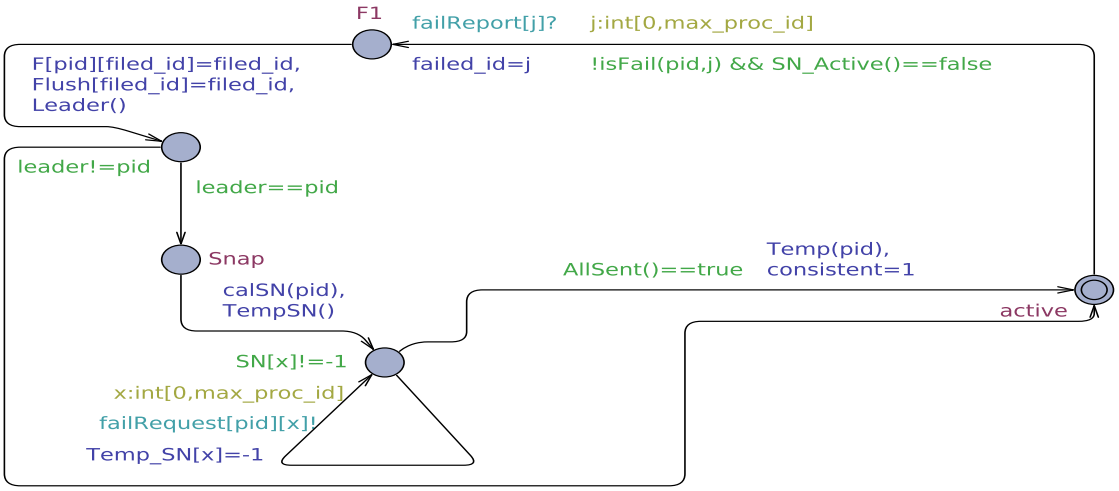

Fig. 5. Formal model for Action F1.

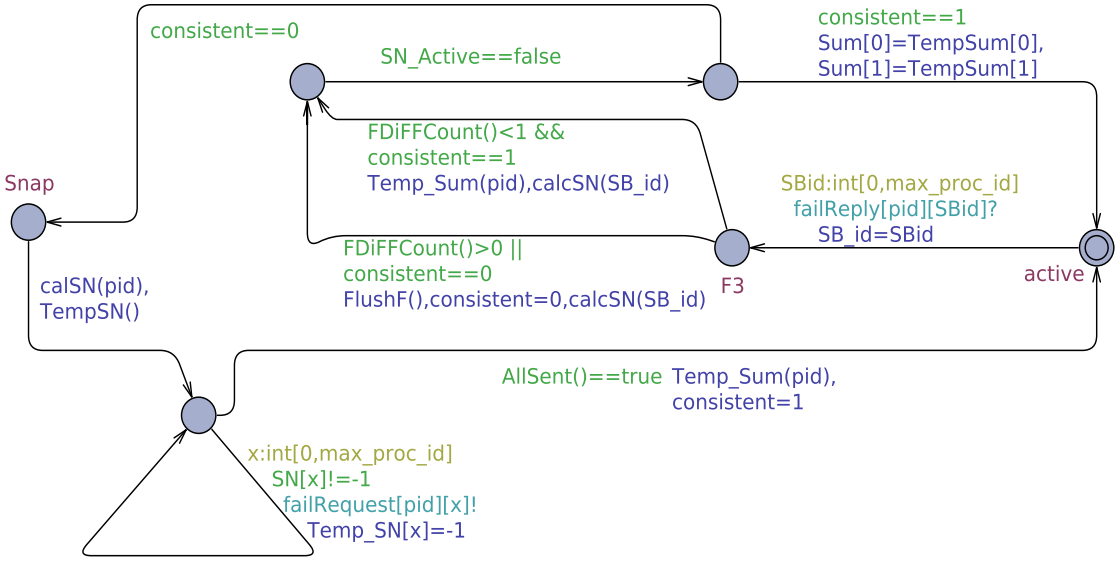

Fig. 6. Formal model for Action F3.

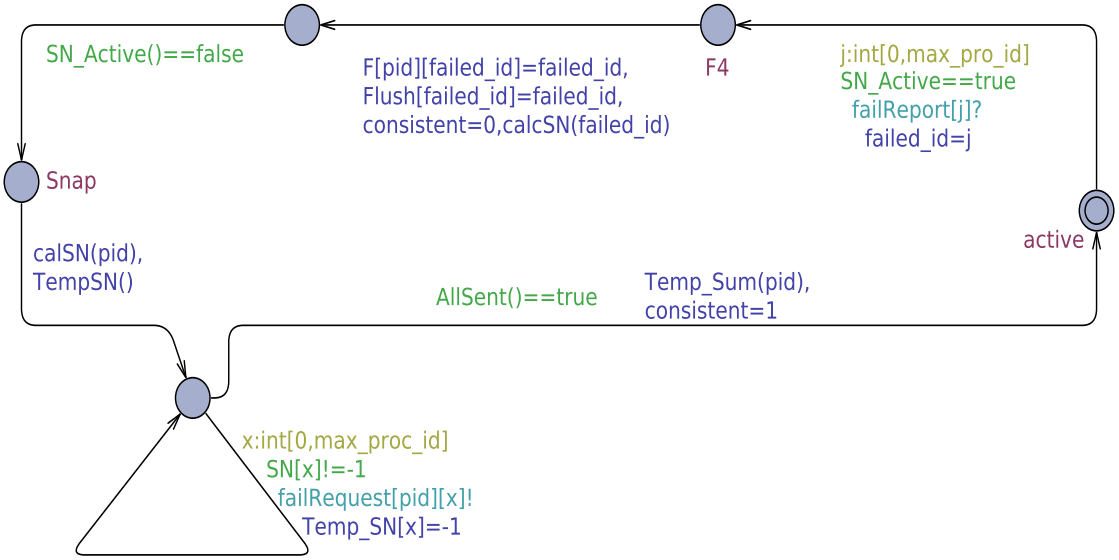

Fig. 7. Formal model for Action F4.

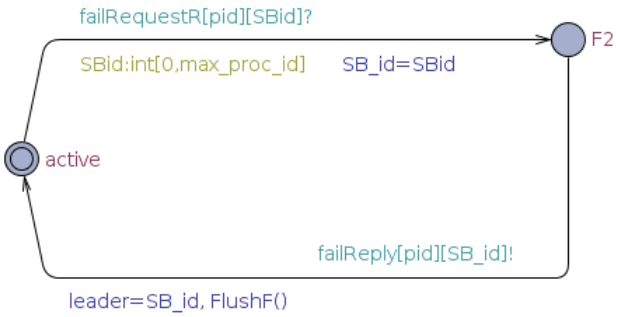

Fig. 8. Formal model for Action F2.

Fig. 9. Formal model for failed process.

TABLE VI. LOCAL DECLARATIONS FOR TERMINATION PROCESS IN MODEL2

leader	This variable stores the id of the current *leader* of the system known to a process. It may be different for each process.
SB_id and *SB_id2*	These variables record the ids of the message sender process at different transitions.
consistent	The boolean variable that shows the consistency of the *snapshot request* sent by the *leader*.
TempSum[]	The *TempSum* array stores the sum of all the weights during the snapshot.
SN[]	The *SN[]* keeps the set of processes to which the *snapshot request* is to be sent by the *leader*.
Temp_SN[maxproc]	The *Temp_SN* array temporarily records the set of processes to which the *snapshot request* message is to be sent by the *leader*. After sending the *snapshot request* to a process, the sender removes this process from *Temp_SN[]*. This array becomes empty after sending *snapshot request* message to all the healthy processes.
failed_id	A local variable that stores the id of the failed process at different transitions.
Flush[]	This array keeps the record of failed processes for which all further communications are flushed.
Stemp[]	The *Stemp* array contains the list of all the instances of *Termination* process taking part in the system.

AddOut(): It calculates the total outgoing weights of all the failed processes known to the current *leader* during the *snapshot*. It then adds the outgoing weights of every failed process to make a sum of outgoing weights in *FOut[]*.

FIin_FOut_Diff(): This function checks the difference of incoming weights and outgoing weights. It uses the *FIn[]* for incoming weights and *FOut[]* for outgoing weights. It subtracts the outgoing weights from incoming weights. Then it moves the difference to *FI_FO_Diff[]*.

Temp_Sum(): This function adds the *FI_FO_Diff* values with *[1/n]*. The sum is stored in *TempSum[]*. The *Temp_Sum()* function calls *FIn_FOut_Diff()* function, the *FIn_FOut_Diff()* function calls *AddOut()* function, the *AddOut()* function calls *AddIn()* function. In this way all the calculations are done properly. The benefit of calling functions inside other functions is that we just call the *Temp_Sum()* function on a transition and all the calculations for *TempSum[]* are done properly.

AllSent(): It checks if the *snapshot requests* have been sent to all the healthy processes.

isFail(): A process uses this function to check the entry of a failed process in *F[]*. If record found then current process can not detect the failure of this process again.

Leader(): The *Leader()* function makes the new *leader* when a process detects failure of some other process. This function is very important because if the *leader* fails then the system needs a new *leader* to collect the weights and send *snapshot request* messages. This function makes the *leader* to a healthy process with minimum id. If the failing process is not the *leader* then this function again selects the previous *leader*.

SN_Active(): System uses this function at different transitions to check if a *snapshot* is already in progress. The function returns a true value if *snapshot* is already in progress otherwise returns a false value.

FDiFFCount(): It calculates the difference of *F[]* of *snapshot reply* sending process and the *F[]* of the *leader* when the *leader* receives the *snapshot reply* message.

D. The Automaton for SnapBuffer Process

The automaton for the *SnapBuffer* process is depicted in Fig. 10. The process is initiated four times to make four instances, each of which is triggered by a certain communication with the *Termination* process. The specification of the *SnapBuffer* process in the protocol has following communicative choices:

1) Receiving a *snapshot request* message from *leader* to store it.
2) Sending a stored *snapshot request* message to a *Termination* process.
3) Receiving a *snapshot reply* message from a *Termination* process to store it.
4) Sending a stored *snapshot reply* message to *leader*.

The process receives a *snapshot request* message through *failRequest?* channel. The guard makes sure that buffer is empty and can receive this message. After receiving this message the *IsEmpty* variable is assigned a false value to show that now buffer is non-empty. The *SnapBuffer* process delivers the stored *snapshot request* message to the recipient through *failRequestR!* channel. The guard makes sure that buffer contains a message for sending. After sending this message the variable *IsEmpty* is set true to show that now buffer is empty again. The *SnapBuffer* process receives a *snapshot reply* message through *failReply?* channel. The guard ensures that buffer is empty and can receive this message. After receiving this message the *IsEmpty2* variable is assigned a false value to show that now buffer is non-empty. The *SnapBuffer* process sends the stored *snapshot reply* message to the *leader*

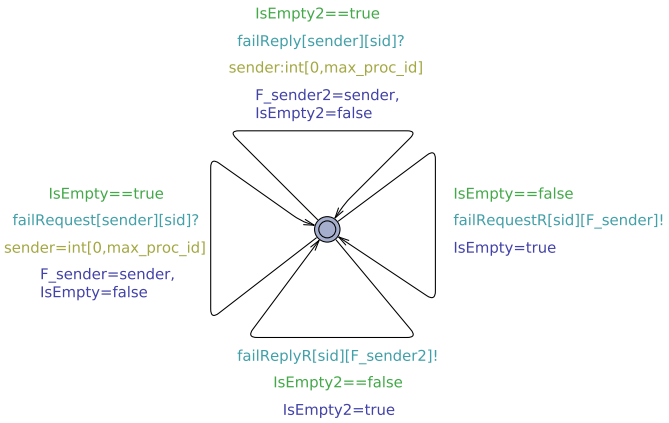

Fig. 10. Formal model for *SnapBuffer* process.

through *failReplyR!* channel. After sending the message the variable *IsEmpty2* is set true to show that now buffer is empty again.

The local declarations of *SnapBuffer* process are described in Table VII.

TABLE VII. LOCAL DECLARATIONS FOR *SnapBuffer* PROCESS

F_sender and *F_sender2*	These variables store the process id of the *snapshot request* sender and process id of the *snapshot reply* sender, respectively.
IsEmpty and *IsEmpty2*	Boolean variables which indicate free space for incoming *snapshot request* message and incoming *snapshot reply* message, respectively.

VI. FUNCTIONAL REQUIREMENTS

The functional requirements illustrate the behaviour of the system and explain what an intended system should do. In other words, they describe the functionality of the system. Every protocol has some functional requirements. We discuss and verify these requirements for both models separately.

A. Functional Requirements for Model1

We extract three functional requirements from the protocol for Model1. These are given as:

R1: No deadlock is supposed to be there except when the *leader* process is at *announce* state and all other processes are at *passive* state. This indicates that all the processes are idle and the *leader* has collected all the weights successfully resulting in proper termination of the system.

R2: This requirement states that after doing certain communications and collecting the weights of other processes, the *leader* process eventually reaches at *announce* state.

R3: According to this requirement, after doing certain message communications all the processes must be idle at *passive* state. All the processes eventually reach at *passive* state.

There are three invariants given in the protocol at page 12 of [7]. These invariants are expected to be preserved by the system.

INV1: In Model1, no process fails. It means all the processes are healthy. The process with minimum id is the *leader* of the system. This invariant is for all the healthy processes other than the *leader*. This invariant states that at any time, if a process is at *passive* state then it must have a zero weight. Similarly, if a process has zero weight then it must be at *passive* state.

INV2: This invariant is related to message sending. All processes can pass *basic* messages to each other. Every process can also send *control* messages to the *leader*. A process sends some weight with *basic* and *control* messages. This invariant describes that the weight sent with any message must be greater than zero.

INV3: At the start of the computation, every process is given an equal initial weight. A process sends some of its weight when it sends a *basic* or *control* message. Each process receives some weight when it receives that message. It updates its weight after sending or receiving a message. All processes also record their incoming and outgoing weights. This invariant states that for all healthy processes at any time, the sum of current weight and all outgoing weights of a process must be equal to the sum of its initial weight and all incoming weights.

B. Functional Requirements for Model2

For Model2, we extract three functional requirements.

R1: This requirement describes that some process reaches at the *announce* state to make sure that all the weights are collected and the system is terminated properly.

R2: A faulty process cannot be the *leader* of the system. This requirement is not satisfied. We discuss a scenario in which this requirement is violated. We have 4 instances of the *Termination* process. These are *p0*, *p1*, *p2* and *p3*. The *p0* is the *leader* of the system. The *leader* sends the *snapshot request* message to all the processes. These messages are yet stored in buffers and not delivered to the recipients. Meanwhile, the *leader* fails. The *p1* detects the *leader* to be faulty and calculates the new *leader* with minimum id from healthy processes. It becomes the *leader* itself. But in future, when it receives the stored *snapshot request* message sent by *p0*, it makes the *p0* as the *leader* of the system. The *p0* is faulty and is supposed to be the *leader* of the system again. That is why this is the clear violation to this functional requirement.

R3: This requirement states that every time the *leader* fails, the *snapshot* is called. The healthy process with minimum id calls the *snapshot*. But this requirement is trivially violated when the *p0* fails and the *p1* becomes passive without detecting the

fault. The *p2* and *p3* also become *passive*. Now the *snapshot* is never called by any process.

The three invariants discussed for Model1 must be preserved for Model2 also.

VII. Formal Specification of Requirements

In this section, we describe formal specification of the requirements and invariants. We also present the formalism for these requirements.

A. Requirement Formal Specification for Model1

According to requirement R1, there should not be a deadlock if the *leader* is not at *announce* state or any of the other process is not at *passive* state. The formula for this requirement is given as.

```
A[] deadlock imply(Termination(0).announce and
        Termination(1).passive and
          Termination(2).passive)
```

The requirement R2 says that the system is terminated properly if the *leader* reaches at *announce* state for every path of execution. The formula for the requirement is given as:

```
A<> Termination(0).announce
```

All the processes must reach at *passive* state for proper termination of the system according to the requirement R3. Given below is the formula for R3:

```
E<> forall(i:id_t) Termination(i).passive
```

The formula for INV1 is presented below. A process moves from *passive* to *A2_1* state after receiving a *basic* message. Then this process updates its weight in next transition. It means, like *passive* state this process has a zero weight at *A2_1* state. So, we are including this state in the formula for INV1.

```
   A[] ((Termination(1).passive imply
    Termination(1).weight[0]==0) and
   (Termination(1).weight[0]==0 imply
Termination(1).passive or Termination(1).
A2_1)) and ((Termination(2).passive imply
    Termination(2).weight[0]==0) and
   (Termination(2).weight[0]==0 imply
Termination(2).passive or Termination(2).
A2_1))
```

Now we discuss the formula for INV2. Each process updates two global variables W1 and W2 after receiving a *basic* or a *control* message. The W1 records the first value and W2 records the second value of received weight. If these variables always keep some non-zero value of weight then it means every message sent by any process has a non zero weight. The formula for this invariant is given as:

```
A[] W1!=0 and W2!=0
```

The invariant INV3 is for all healthy processes. The formula for this invariant is:

```
A[] forall(i:id_t)
Termination.In_Out_Equal(i)==true
```

This invariant INV3 uses five functions for its calculations. These functions are *AddInWeights()*, *InSum()*, *AddOutWeights()*, *OutSum()*, and *In_Out_Equal()*. All these functions perform combined functionality for verification of INV3. The function *AddInWeights()* adds all the incoming weights recorded in *In_arr[]* of calling process. The function *InSum()* adds the current weight and the some of incoming weights and stores the result in *tempIn[]*. The *AddOutWeights()* function adds all the outgoing weights recorded in *Out_arr[]* of a process. The function *OutSum()* calculates the sum of initial weight and all outgoing weights of a process and stores the result in *tempOut[]*. At the end *In_Out_Equal()* checks the equality of *tempIn[]* and *tempOut[]*. If both arrays are equal then this function returns a true value otherwise a false value.

B. Requirement Formal Specification for Model2

According to requirement R1, some process reaches at the *announce* at some time. The formula for this requirement is given as:

```
A<> exists(i:id_t) Termination(i).announce
```

The requirements R2 and R3 are clearly discussed with examples in Model2 requirements part. Now we discuss the formula for invariant INV1. Formalism for this invariant is given as:

```
A[] forall(i:id_t((Termination(i).notMin(i)==
 true and Termination(i).passive imply
   Termination(i).weight[0]==0) and
  (Termination(i).notMin(i)==true and
   Termination(i).weight[0]==0 imply
Termination(i).passive or Termination(i).A2_1))
```

The function *notMin()* checks if the calling process belongs to faulty set of processes or it is a healthy process with minimum id. It returns false if the process is faulty or it is healthy with minimum id. It returns true otherwise allowing other processes to check their weight at the *passive* state.

The formalism for INV3 for Model2 is similar to formalism of INV3 for Model1. All the functions and their definitions are same. But Model2 uses an extra function *Equal()* that checks if the calling process is faulty. It means we are just concerned with the calculations for healthy processes. If calling process is healthy then it returns true if the invariant is preserved and returns false if the invariant is violated. The formula for this invariant is given as:

```
A[] forall(i:id_t) Termination(i).Equal(i)==
true
```

VIII. Verification Results for Model1

This section shows the simulation results of formalism for functional requirements and invariants for Model1. These results are collected by executing the formulas in verifier of the UPPAAL toolset. For simplicity we use the *Buffer* instead of the *MessageBuffer* in all counterexamples. Results for Model1 are given below in Table VIII. We verify our system model for,

Total Number of *Termination* Process Instances = 3

Total Weight of the System = 1

TABLE VIII. REQUIREMENT RESULTS FOR MODEL1

Requirement	Status	Computational Time
R1*	Satisfied, 225504590 states	57600m23.345s
R2*	Satisfied, 32045480 states	7200m53.167s
R3	Satisfied, 18986 states	3.564s
INV1*	Satisfied, 28006031 states	7210m34.453s
INV2*	Satisfied, 25933265 states	7206m12.560s
INV3*	Satisfied, 20568945 states	7002m19.350s

Weight of Each Instance = 1/3

Requirement R3 is satisfied. For R1, R2, INV1, INV2 and INV3, state space is not fully explored. We are not sure about their final results. We explored these requirements for breadth first searching technique. Our claims are limited to the number of states and time given in Table VIII.

IX. VERIFICATION RESULTS FOR MODEL2

Results for Model2 are given below in Table IX. We verify our system model for,

Total Number of *Termination* Process Instances = 4

Total Weight of the System = 1

Weight of Each Instance = 1/4

TABLE IX. REQUIREMENT RESULTS FOR MODEL2

Requirement	Status	Computational Time
R1	Not satisfied	06.307s
INV1*	Satisfied, 25166435 states	7215m33.873s
INV2*	Satisfied, 42542339 states	11520m21.212s
INV3	Not satisfied	20.353s

In Model2, for INV1 and INV2, state space is not fully explored. We are not sure about the final results of these invariants. We explored these requirements using breadth first searching technique and we claim these results to the number of states given in Table IX. The requirement R1 is not satisfied. The counterexample for this violation is shown in Fig. 11. The *p3* sends a *basic* message to *p0*. The *p1* and *p2* send *control* messages to *p0*. Then *p0* sends a *basic* message to *p3*. After this, *p3* receives the *basic* message sent by *p0* and sends the *control* message to *p0*. After doing all these communications, *p0* fails and other three processes become *passive*. In this case, there is no process that is *active* and detect *p0* as faulty. So, it is not possible to collect the weights carried by *p0*. Therefore, the *announce* state is not reachable in this scenario.

The invariant INV3 violates in the given scenario. The *p0* sends a *basic* message to *p1* and becomes *passive*. The *p1* sends a *basic* message to *p0* and reaches at *A1* state. The *p2* sends a *control* message to *p0* and visits *A3* state. After this, *p0* receives the *control* message of *p0* and becomes *passive*. At *passive* state, it receives the *basic* message from *p1* and reaches at *active* state. In the whole activity *p3* stays at *active* state. The invariant is not satisfied in this situation. The counter example for this violation is shown in Fig. 12.

X. LIMITATIONS AND CHALLENGES

There are some limitations for verification of intended termination detection protocol. We limit the concurrent *termination* processes to three in case of fault-free distributed system and four for faulty distributed system. The *basic* message sending limit for any process is two. The models generate a huge state space with millions of states. The purpose of these limitations is to reduce the state space of our computations. The machines and servers used in our verification have limited resources for memory and speed. We performed some computations on the machine with 16GB RAM, core i7(4th Gen) CPU and 3.4 GHz clock speed. We also performed some computations on mammoth server (tue.nl) which has 56 machines with 2 GHz speed and a total of 935GB RAM.

We faced some challenges during the modeling and verification of the protocol. The total weight of the system is 1 and initial weight of each process is *1/n* where *n* is the total number of processes. The expression *1/n* returns fractional values. At the end of the computation it becomes hard to make the sum of all weights to 1 due to possible rounding errors. The first challenge was to manage a single weight in the form of a pair of integers. We presented the weight *1/n* in the form of two values as *[1,n]*. This method created many difficulties for accurately calculating the incoming, outgoing and current weights of the processes. This was also a big challenge to correctly model the protocol and its invariants in UPPAAL. We used model abstraction for reducing the state space that was also a challenge.

XI. CONCLUSION

We formalised both parts of termination detection protocol as specified in [7] in the timed-automata-theoretic formalism of UPPAAL. We formally specified and performed verification analysis of the protocol with respect to its functional requirements and invariants. During our formal analysis, we found some scenarios in which the protocol does not meet its functional requirements. Counter example are there to witness the claimed results.

REFERENCES

[1] W. Fokkink, *Distributed algorithms: an intuitive approach*. MIT Press, 2013.

[2] K. M. Chandy and J. Misra, "Asynchronous distributed simulation via a sequence of parallel computations," *Commun. ACM*, vol. 24, no. 4, pp. 198–206, Apr. 1981. [Online]. Available: http://doi.acm.org/10.1145/358598.358613

[3] K. M. Chandy, J. Misra, and L. M. Haas, "Distributed deadlock detection," *ACM Trans. Comput. Syst.*, vol. 1, no. 2, pp. 144–156, May 1983. [Online]. Available: http://doi.acm.org/10.1145/357360.357365

[4] G. Tel and F. Mattern, "The derivation of distributed termination detection algorithms from garbage collection schemes," *ACM Trans. Program. Lang. Syst.*, vol. 15, no. 1, pp. 1–35, Jan. 1993. [Online]. Available: http://doi.acm.org/10.1145/151646.151647

[5] E. W. Dijkstra and C. Scholten, "Termination detection for diffusing computations," *Information Processing Letters*, vol. 11, no. 1, pp. 1–4, 1980. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0020019080900216

[6] N. Francez, "Distributed termination," *ACM Trans. Program. Lang. Syst.*, vol. 2, no. 1, pp. 42–55, Jan. 1980. [Online]. Available: http://doi.acm.org/10.1145/357084.357087

[7] Y. Tseng, "Detecting termination by weight-throwing in a faulty distributed system," *J. Parallel Distrib. Comput.*, vol. 25, no. 1, pp. 7–15, 1995. [Online]. Available: https://doi.org/10.1006/jpdc.1995.1025

[8] X. Wang and J. Mayo, "A general model for detecting distributed termination in dynamic systems," in *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, April 2004, p. 84.

[9] G. Tel and F. Mattern, "The derivation of distributed termination detection algorithms from garbage collection schemes," *ACM Trans. Program. Lang. Syst.*, vol. 15, no. 1, pp. 1–35, Jan. 1993. [Online]. Available: http://doi.acm.org/10.1145/151646.151647

[10] F. Mattern, H. Mehl, A. A. Schoone, and G. Tel, "Global virtual time approximation with distributed termination detection algorithms," Tech. Rep., 1991.

[11] S. Chandrasekaran and S. Venkatesan, "A message-optimal algorithm for distributed termination detection," *J. Parallel Distrib. Comput.*, vol. 8, no. 3, pp. 245–252, mar 1990. [Online]. Available: http://dx.doi.org/10.1016/0743-7315(90)90099-B

[12] J. Pang, *Analysis of a Security Protocol in μCRL*, C. George and H. Miao, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002. [Online]. Available: http://dx.doi.org/10.1007/3-540-36103-0_40

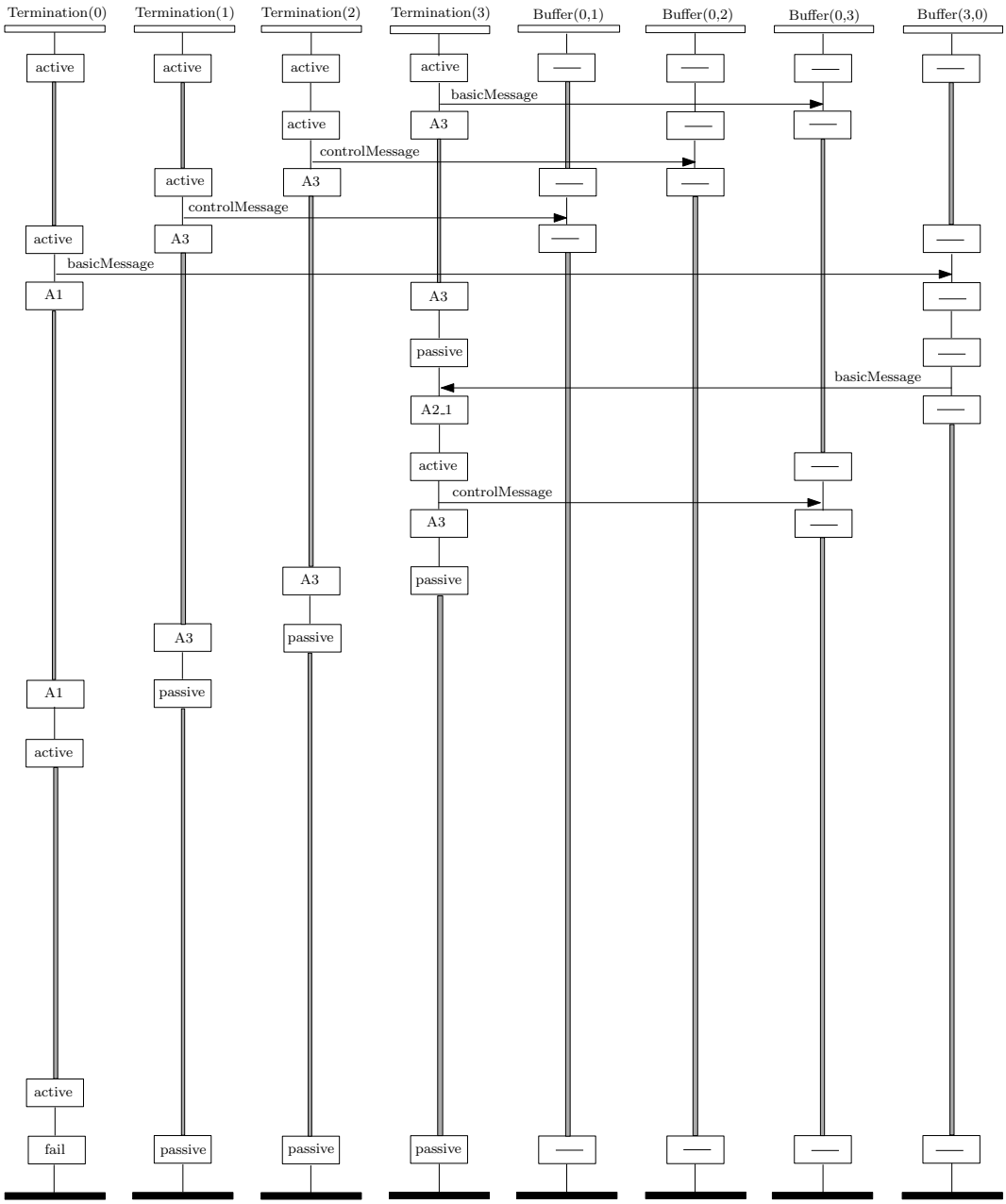

Fig. 11. Trace for R1 in Model2.

[13] W. H. J. Feijen and A. J. M. van Gasteren, *Shmuel Safra's Termination Detection Algorithm*. New York, NY: Springer New York, 1999. [Online]. Available: http://dx.doi.org/10.1007/978-1-4757-3126-2_9

[14] N. Mittal, S. Venkatesan, and S. Peri, *Message-Optimal and Latency-Optimal Termination Detection Algorithms for Arbitrary Topologies*, R. Guerraoui, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30186-8_1

[15] N. R. Mahapatra and S. Dutt, "An efficient delay-optimal distributed termination detection algorithm," *Journal of Parallel and Distributed Computing*, vol. 67, no. 10, pp. 1047–1066, 2007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731507000998

[16] J. S. Ostroff, "Formal methods for the specification and design of real-time safety critical systems," *J. Syst. Softw.*, vol. 18, no. 1, pp. 33–60, Apr. 1992. [Online]. Available: http://dx.doi.org/10.1016/0164-1212(92)90045-L

[17] A. O. Gomes and M. V. M. Oliveira, *Formal Specification of a Cardiac Pacing System*, A. Cavalcanti and D. R. Dams, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-05089-3_4

[18] Y. K. H. Lau, "The design of distributed safety critical software using csp," in *IEE Colloquium on Safety Critical Software in Vehicle and Traffic Control*, Feb 1990, pp. 8/1–8/5.

[19] K. G. Larsen, P. Pettersson, and W. Yi, "Uppaal in a nutshell," 1997.

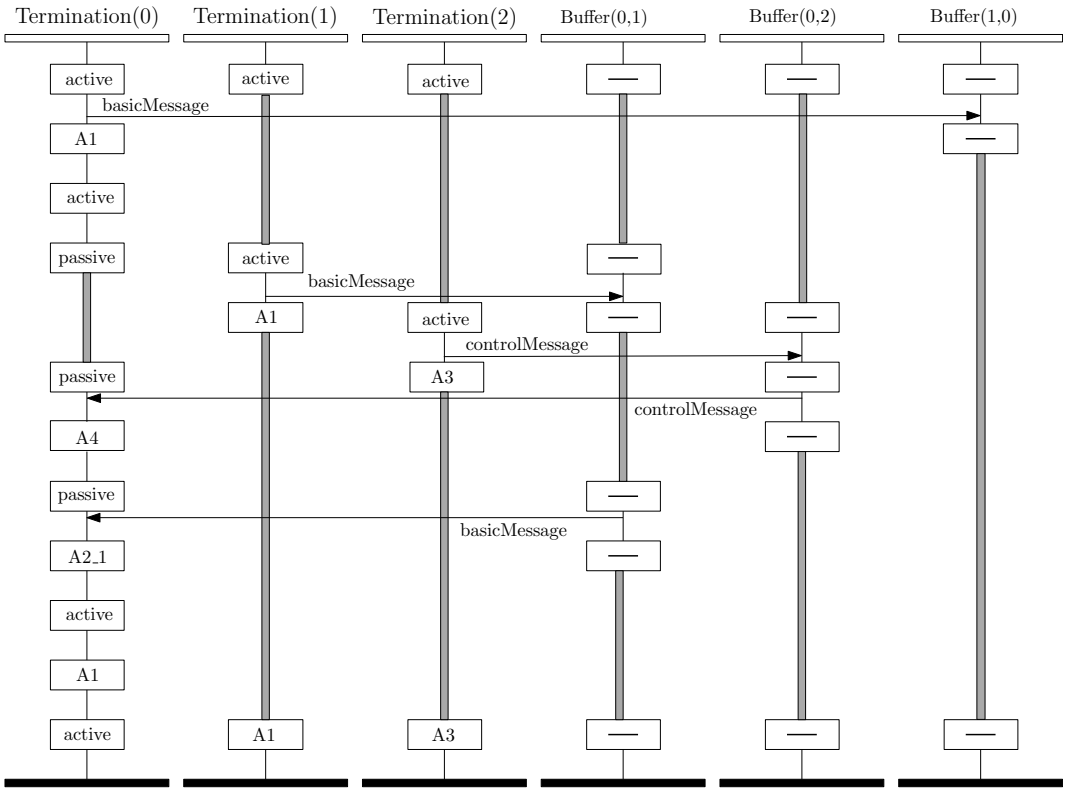

Fig. 12. Trace for INV3 in Model2.