Preprint: Published in "Concurrent Programming, Open Systems and Formal Methods: Essays Dedicated to Prof. Gul Agha to Celebrate his Scientific Career," José Meseguer, Carlos A. Varela, and Nalini Venkatasubramanian, Eds., Springer, LNCS 16120, DOI: 10.1007/978-3-032-05291-9_8, pp 189-214, September 2025

Actors for Timing Analysis of Distributed Redundant Controllers

Marjan Sirjani[1 [0000−0001−5478−0987]], Edward A. Lee[2 [0000−0002−5663−0584]], Zahra Moezkarimi[1 [0000−0001−5495−9098]], Bahman Pourvatan[1 [0009−0000−6330−0412]], Bjarne Johansson[1,3 [0000−0002−5333−3699]], Stefan Marksteiner[1,4 [0000−0001−8556−1541]], and Alessandro V. Papadopoulos[1 [0000−0002−1364−8127]]

[1] Mälardalen University, Västerås, Sweden
marjan.sirjani@mdu.se, zahra.moezkarimi@mdu.se, bahman.pourvatan@mdu.se, stefan.marksteiner@mdu.se, alessandro.papadopoulos@mdu.se
[2] University of California at Berkeley, Berkeley, USA
eal@berkeley.edu
[3] ABB AB, Västerås, Sweden
bjarne.johansson@se.abb.com
[4] AVL List Gmbh, Graz, Austria
stefan.marksteiner@avl.com

Abstract. We use two actor-based languages, Timed Rebeca and Lingua Franca, to show modeling, model checking, implementation, and timing analysis of an industry-suggested algorithm for role selection in distributed control systems with redundancy. The algorithm prioritizes consistency over availability in trade-off situations. We show scenarios that simulate the environment and possible faults and use the Timed Rebeca model checking tool to investigate whether they may cause a failure. We also show the maximum latency that can be tolerated without causing inconsistency. We then use the coordination language Lingua Franca to implement the model. It can also simulate network switches, allowing you to set up test scenarios that include network degradation, such as switch failures, packet losses, and excessive latency. This can be set up as a hardware-in-the-loop simulation, where the actual node implementations interact with simulated switches and the network.

Prologue

To honor the profound impact of Gul Agha's visionary contributions to concurrent computation and the actor model, we dedicate this work to him with deep admiration. On a personal note, the first author expresses heartfelt gratitude for Gul's deep wisdom, vast knowledge, peaceful manner, and warm, unwavering support—qualities that have left a lasting impression both intellectually and personally.

1 Introduction

The actor model is among the pioneering ones in addressing concurrent and distributed applications. The model was originally introduced by Hewitt in the 70s as a formalism for artificial intelligence and an agent-based language for programming distributed systems [7, 8, 5]. The actor model is then promoted as a model of concurrent computation in distributed systems in Gul Agha's PhD thesis in 1985 [3] and as a conceptual foundation for concurrent object-oriented programming [2, 1]. The actor model has been used both as a framework for theoretical understanding of concurrency and as the basis for several practical implementations of concurrent and distributed systems.

Rebeca is introduced as an actor-based language with model checking support in 2001 by Sirjani et al. [29, 30], and the timed version is introduced in 2007 [25, 14, 26]. The language design follows the principle to keep the core language as simple as possible and suitable for analysis [24]. Rebeca is imperative rather than functional, and actors are units of concurrency, meaning there is no intra-actor concurrency. Lingua Franca is an actor-based language introduced in 2019 [18, 19] based on the reactors coordination model first presented by Lohstroh et al. [20]. The reactor model combines several ideas to enable determinism while preserving much of the style of actors. The reactor model uses timestamps to make programs deterministic by default and uses a logical model of time that can be associated with physical time to provide control over timing during execution. Timed Rebeca is extended to support priorities in execution of actors and handlers of actors to adopt the Lingua Franca's deterministic order of execution when there is more than one reaction enabled at the same logical time [28, 27, 15].

The timestamps introduced in Timed Rebeca and in Lingua Franca introduce some significant differences compared to Hewitt-Agha actors. The programs become more deterministic, particularly with the use of priorities in Timed Rebeca, but they also introduce the possibility of unfairness. In particular, it is possible to create programs that exhibit Zeno conditions, meaning that there will be an infinite number of events with timestamps below some finite threshold. Such programs may be unfair because events with timestamps larger than the finite threshold will never be processed. Also, although dynamic communication topologies is supported by both Timed Rebeca and Lingua Franca, we do not consider it here. It is not needed for the applications we consider. Dynamic actor creation is possible in LF, but again, not needed here. It is not supported by Timed Rebeca because it makes formal verification considerably more difficult.

In this paper, we use an industry-suggested algorithm for redundancy role selection in distributed control systems with redundancy to demonstrate how we use Timed Rebeca and Lingua Franca for timing analysis for a distributed and concurrent system where timing is crucial and can change the outcome. The Network Reference Point Failure Detection (NRP FD) algorithm is designed to detect failures between redundant controllers [11]. The algorithm prioritizes consistency over availability in tradeoff situations but minimizes the possibility of lack of availability. We model-checked the NRP FD algorithm in [10], checked the

situations that cause dual primaries, and generated test cases. Here, we show the time analysis to find the minimum time between failures that can cause double primaries.

In distributed applications, Brewer's CAP theorem tells us that when networks become partitioned (P), one must give up either consistency (C) or availability (A). Consistency is agreement on the values of shared variables; availability is the ability to respond to reads and writes, accessing those shared variables [6]. Lee et al. [17] explain that in real-time systems, the time it takes for a software subsystem to respond through an actuator to a stimulus from a sensor is a critical property of the system. They, therefore, generalize the notion of availability to include this time, not just the system's response time to human users. A software subsystem where sensor-to-actuator response time is large is less available than one for which it is small. They argue that availability, a real-time property of a system, and consistency, a logical property, relate numerically to clock synchronization and latencies introduced by networks and computation. They replace the P in CAP with L, representing latencies that include execution times, network delays, and clock synchronization errors. The CAP theorem becomes the CAL theorem, which gives an algebraic relationship between the three quantities: consistency, availability, and latency.

In the following, we will briefly overview Timed Rebeca and Lingua Franca languages and explain the NRP FD algorithm. Then we show how NRP FD is modeled in Timed Rebeca and how model checking the Timed Rebeca model can help for time analysis of the algorithm. Using model checking, we can find the combinations of timing configurations of the controllers and switches (including heartbeat and ping periods and network delays) and the time between fault occurrences that can cause inconsistencies. We then use Lingua Franca to implement and simulate these situations.

2 Actor-based Languages, Timed Rebeca and Lingua Franca

Rebeca (Reactive Object Language) [29, 23] is an actor-based language designed to model and formally verify reactive concurrent and distributed systems. In Rebeca models, reactive objects known as rebecs resemble actors with no shared variables, asynchronous message passing, and unbounded message buffers. Each rebec has a single thread of execution. Communication with other rebecs is achieved by sending messages, and periodic behavior is executed by sending messages to itself. Rebeca has no explicit receive statement, and its send statements are non-blocking. Each rebec has variables, methods (message servers), and a dedicated message queue for received messages. How a rebec reacts to a message is specified in message servers. The rebec processes messages by de-queuing from the top and executing the corresponding message server non-preemptively. The state of a rebec can change during the execution of its message servers through assignment statements.

Rebeca is an imperative language with syntax similar to Java. A Rebeca model consists of several reactive classes and a main section. Each reactive class describes the type of a certain number of rebecs. Rebecs (actors) are instantiated in the main block. While message queues in the semantics of Rebeca are inherently unbounded, a user-specified upper bound for the queue size is necessary to ensure a finite state space during model checking. Reactive classes include constructors, which share the same name as the class. These constructors are responsible for initializing the actor's state variables and placing initially required messages in the actor's message buffer.

In this work, we use Timed Rebeca (the timed extension of Rebeca) [26, 14] with a global logical time. Timed Rebeca considers synchronized local clocks for all actors throughout the model. Instead of a message queue, Timed Rebeca uses a message bag in which messages carry their respective time tags. The sender tags its local time to a message when sending. Timed Rebeca introduces three timing primitives: "delay," "after," and "deadline." A delay statement represents the passage of time for an actor while executing a message server, i.e., it is used to model computation times. All other statements are assumed to execute instantaneously. The keywords "after" and "deadline" are augmented to a message send statement. The term "after(n)" means it takes n units of time for a message to reach its receiver. Using the after construct, we can model network delay and periodic events. We can use a nondeterministic assignment to n and model nondeterministic arrival times for a message (event). The term "deadline(n)" conveys that if the message is not retrieved within n units of time, there will be a timeout. An abstract syntax of Timed Rebeca is provided in Appendix A. Timed Rebeca is extended with priorities [28]. Priorities are assigned to rebecs and message handlers to control the order of their execution and hence enhance the determinism of the system's behavior [15]. If more than one actor or event is enabled at the same logical time, the model checker builds all the possible execution traces. Using priorities, you can cut some of the branches or even completely control the order in which events are handled at that logical time.

Lingua Franca (LF) [20, 18] is a coordination language designed for embedded real-time systems. Software components are called "reactors." The messages exchanged between reactors have logical timetags drawn from a discrete, totally ordered model of time. Every reactor will react to incoming messages in timetag order. The Lingua Franca compiler ensures that all logically simultaneous messages are processed in precedence order, making the computation deterministic even with parallel execution.

The Lingua Franca code consists of a set of reactors and a main reactor. Reactors contain state variables, input and output ports, and physical actions and reactions. The body of reactions can be written in the target language supported by LF, including C, C++, and TypeScript. In each case, the LF compiler generates a standalone executable in the target language. A reactor may also react to a "physical action," typically triggered by some external event

such as a sensor. The physical action will be assigned a timetag based on the current physical clock on the machine hosting the reactor.

Lingua Franca includes a notion of a deadline, a relation between logical time and physical time, as measured on a particular platform. Specifically, a program may specify that the invocation of a reaction must occur within some physical time interval of the logical timestamp of the message. This, together with physical actions, can be used to ensure some measure of alignment between logical time and some measurement of physical time.

3 Industrial Controller Redundancy and NRP FD Algorithm

Industrial controllers are robust and specialized computers designed to execute control applications in a deterministic and reliable manner. These controllers form the foundation of automation solutions across diverse domains, such as propulsion and energy generation automation on ships and ventilation automation in traffic tunnels. They are also crucial in domains like offshore oil extraction platforms, where downtime is highly undesirable due to safety and economic considerations [10]. To enhance reliability in such systems, single points of failure in the automation infrastructure, including controllers, are mitigated through redundancy. Controller redundancy typically implies a standby redundancy scheme, where one controller functions as the active primary while a secondary remains on standby [22]. These controllers, or Distributed Controller Nodes (DCNs), manage processes via Field Communication Interfaces (FCIs) connected to input/output (I/O) devices interfacing with the physical environment. The controllers monitor the state of the physical process by sampling values from the I/O devices through the FCI. Based on these readings, they compute appropriate output values to drive the process toward a desired state. The standby controller requires regular updates on the state from the primary and heartbeat signals; consequently, effective communication between the controllers is essential.

In case of communication failure between the redundant controllers, two strategies are available: (i) disabling redundancy when only one communication path remains or (ii) continuing in redundant mode. These strategies align with the CAP theorem. Disabling redundancy maintains consistency but sacrifices availability, while continuing in redundant mode risks consistency if the failure of the final link is indistinguishable from the failure of the active controller. The NRP FD algorithm is designed to detect failures between a redundant controller pair while preserving consistency and minimizing the availability tradeoff. In a typical standby redundancy setup, one controller acts as the active primary and the other as the standby backup. Only the primary controller provides output to the FCI and connected I/O devices to maintain consistency. Upon detecting a primary controller failure, the backup assumes the primary role. For this synchronization and role determination, communication between the controllers, typically over a switched network, is vital [4, 16]. The controllers are often con-

nected via dual, independent network paths, forming a redundant network (see Figure 1). NRP FD is a push-based failure detection algorithm. The primary controller sends heartbeat messages at fixed intervals to the backup over the redundant network [11]. Using a Network Reference Point (NRP) distinguishes NRP FD from other heartbeat-based algorithms. The NRP serves as a reference on the network, typically a switch, and must meet two criteria: (i) it should not share any common cause failure with the DCNs, and (ii) it should be accessible by at most one DCN in the event of network partitioning. Each DCN maintains a list of NRP candidates for additional robustness. If the designated NRP fails while communication paths remain functional between the redundant pair, the primary can propose an alternative NRP candidate to the backup.

In Figure 1, the upper network comprises switches `SwitchA1`, and `SwitchA2`, `SwitchA3`. The primary controller's NRP candidates are `SwitchA1` and `SwitchB1`, and the backup's are `SwitchA3` and `SwitchB3`, with the current NRP set to `SwitchA1`.

The operational behavior of NRP FD can be summarized as follows: the primary controller selects an NRP from its candidates and communicates this choice in the heartbeat messages. It continually monitors its ability to reach the NRP. If the NRP becomes unreachable, the primary proposes a new NRP to the backup. The primary forfeits the primary role if the backup does not acknowledge the proposed change within a predefined timeframe. Meanwhile, the backup monitors heartbeats from the primary. If no heartbeat is received within a set interval, the backup verifies its ability to reach the NRP. If it can reach the NRP, the backup assumes the primary role.

4 Modeling of NRP FD using Timed Rebeca

For modeling and verification of the NRP FD, we use the Timed Rebeca and its integrated model checker tool, Afra. For modeling, we have followed the protocol specifications for NRP FD in [11] and refined the model based on discussions with industrial partners to ensure accuracy and relevance. This version enhances the previous model provided in [9] by making overall improvements, removing unnecessary conditions, and modifying the message handling flow. These changes

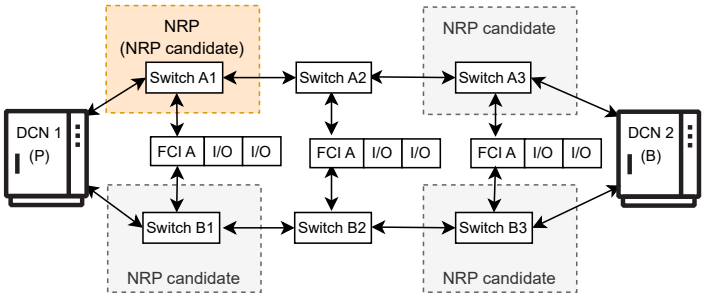

Fig. 1: A redundant network with the NRP and NRP candidates highlighted.

result in a model that more accurately reflects real-world behavior. The complete updated model can be found in Appendix B and also on the Rebeca GitHub page[5].

In the Timed Rebeca model, each node and switch is modeled as an actor. They communicate by message passing. A Rebeca model consists of reactive class definitions that dictate the behavior of rebecs (actors) in the system. Their responses to messages (and timed events) are handled by message servers. L1[6] illustrates selected parts of the Timed Rebeca model for NRP FD. The model is extensible, allowing an increase in the number of switches and nodes.

The NRP-FD model has two main element types: Nodes and Switches. Each of these element types is defined as a reactive class, with `Node` (L1, line 9) and `Switch` (L1, line 40). Each reactive class has a constructor, which is a unique method that runs when an object (rebec) is instantiated. This constructor initializes the variables. In the main section (L1, lines 60-70), we instantiate two nodes (`DCN1` and `DCN2` with ids 100 and 101) and six switches (`switchA1-switchA3` and `switchB1-switchB3`). A node can be either a primary node or a backup node. A switch can be either terminal (connected to a DCN) or non-terminal (not directly connected to a DCN). A switch can also serve as an NRP candidate, an NRP, or none of these roles. For each network, each node has an NRP candidate switch. For `DCN1`, the candidates are `switchA1` (id 1) and `switchB1` (id 4). For `DCN2`, the candidates are `switchA3` (id 3) and `switchB3` (id 6) (L1, lines 68-69). The parameters used when instantiating these objects are used to define their roles and pass important information to the constructor.

At the start of the algorithm, `DCN1` (id 100) is set as the primary node in the instantiation (second parameter in lines 68-69 of L1). In the `Node` reactive class, there are two known rebecs, meaning the node can send messages to them. The constructor of `Node` calls the `runMe` message server (L1, line 21). In `runMe` (L1, line 28), the DCN checks its current state using the mode variable and executes the corresponding behavior (L1, lines 30-34). At the end of `runMe` (L1, line 35), the method calls itself using "`runMe() after(heartbeat_period);`", making it a periodic event. This ensures that `runMe` runs at every heartbeat interval, as defined by `heartbeat_period` in the code (L1, line 1). The heartbeat period must be significantly larger than other timing parameters to process all events within a single heartbeat cycle. The choice of timing parameters is carefully considered to make the model as realistic as possible. More details on timing will be discussed in the following.

In NRP FD, DCNs operate in four modes, including `WAITING`, `BACKUP`, `PRIMARY`, and `FAILED`, as shown in Figure 2. The behavior of the nodes based on their roles are defined in the message server `runMe`. Listing 2 (L2), presents key parts of the message server `runMe`. From this point forward, both Figure 2 and the Listing 2 can be followed together for a better understanding.

In the Rebeca model, each DCN starts in the `WAITING` mode, which is set in the `Node` constructor (L1, line 18). The primary id (`Myprimary`) is passed to both

[5] https://rebeca-lang.org/allprojects/DistributedControllers
[6] We use L1 and L2 to refer to Listing 1 and Listing 2, respectively.

```
env int heartbeat_period = 20;
env int max_missed_heartbeats = 2;
env int ping_timeout =10;
env int nrp_timeout = 10;
env byte NumberOfNetworks = 2;
env int switchA1failtime = 0;
...
env int networkDelay = 1;
reactiveclass Node (20){
    knownrebecs {Switch out1, out2;}
    statevars {...}
    Node (int Myid, int Myprimary, int NRPCan1_id, int NRPCan2_id, int myFailTime) {
        id = Myid;
        NRPCandidates[0] =NRPCan1_id;
        NRPCandidates[1] =NRPCan2_id;
        NRP_network = -1;
        primary = Myprimary;
        mode = WAITING;
        ...
        if(myFailTime!=0) nodeFail() after(myFailTime);
        runMe();
    }
    msgsrv new_NRP_request_timed_out(){...}
    msgsrv ping_timed_out() {...}
    msgsrv pingNRP_response(int mid){...}
    msgsrv new_NRP(int mid,int prim, int mNRP_network, int mNRP_switch_id) {...}
    msgsrv request_new_NRP(int origin) {...}
    msgsrv runMe(){
        if(?(true,false)) nodeFail();
        switch(mode){
            case 0: //WAITING : ...
            case 1: //PRIMARY : ...
            case 2: //BACKUP : ...
            case 3: //FAILED : ...
        self.runMe() after(heartbeat_period);
     }
     msgsrv heartBeat(byte networkId, int senderid) {...}
     msgsrv nodeFail(){...}
}
reactiveclass Switch(10){
    knownrebecs {...}
    statevars {...}
    Switch (int myid, byte networkId, boolean term, Switch s1, Switch s2, int
    ↪  myFailTime, Node n1) {
       mynetworkId = networkId;
       id = myid;
       terminal=term;
       amINRP = false;
       failed = false;
       switchTarget1 = s1;
       switchTarget2 = s2;
       ...
    }
    msgsrv switchFail(){ failed = true; amINRP=false;}
    msgsrv pingNRP_response(int senderNode){...}
    msgsrv request_new_NRP(int senderNode) {...}
    msgsrv pingNRP(int switchNode, int senderNode, int NRP) {...}
    msgsrv new_NRP(int senderNode, int mNRP_network, int mNRP_switch_id) {...}
    msgsrv heartBeat(byte networkId, int senderNode) {...}
}
main {
    @Priority(1) Switch switchA1():(1, 0, true , switchA2 , switchA2 ,
    ↪  switchA1failtime, DCN1);
    @Priority(1) Switch switchA2():(2 ,0, false , switchA1 , switchA3 ,
    ↪  switchA2failtime, null);
    @Priority(1) Switch switchA3():(3, 0, true , switchA2 , switchA2 ,
    ↪  switchA3failtime, DCN2);
    @Priority(1) Switch switchB1():(4, 1, true , switchB2 , switchB2 ,
    ↪  switchB1failtime, DCN1);
    @Priority(1) Switch switchB2():(5, 1, false , switchB1 , switchB3 ,
    ↪  switchB2failtime, null);
    @Priority(1) Switch switchB3():(6, 1, true , switchB2 , switchB2 ,
    ↪  switchB3failtime, DCN2);

    @Priority(2) Node DCN1(switchA1, switchB1):(100, 100, 1, 4, node1failtime);
    @Priority(2) Node DCN2(switchA3, switchB3):(101, 100, 3, 6, node2failtime);
}
```

Listing 1: (L1) An abstracted version of the Timed Rebeca model of NRP FD (The full version is in Appendix B).

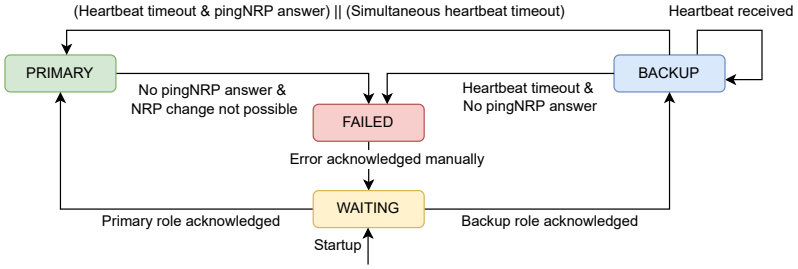

Fig. 2: Different modes of a DCN in NRP FD in the Rebeca model. The initial mode is `WAITING`, from which the node transitions to either `PRIMARY` or `BACKUP`, depending on the value passed to its constructor. From `PRIMARY`, the node moves to `FAILED` if, after sending a `pingNRP`, it does not receive a response from the NRP within the deadline and is unable to switch to another NRP. In the `BACKUP` mode, the node transitions to `PRIMARY` if the heartbeat times out and `pingNRP` detects a reachable NRP, or if the heartbeat timeout occurs simultaneously on both networks. In the latter case, the `BACKUP` node assumes that the `PRIMARY` node has failed, as simultaneous failures in both networks are unlikely. The node remains in `BACKUP` mode as long as it continues receiving heartbeats. If heartbeats stop and there is no response to the ping of the NRP, it transitions to `FAILED` and remains in that mode until it is manually restored.

nodes in the instantiation, and when a node is in `WAITING` mode, it determines its role based on this id. If it is the designated primary, it transitions accordingly, and an NRP is announced (L2, lines 4-11); otherwise, it transits to the `BACKUP` mode (L2, line 12).

In `PRIMARY` mode, the primary DCN checks if the NRP is reachable by sending a `pingNRP` message. The `pingNRP` message is sent based on the network where the NRP is located. This message is handled by the `pingNRP` message server (L1, line 56). In a real system, `pingNRP` could be implemented using an Internet Control Message Protocol (ICMP) echo request (ping) or another suitable protocol based on the NRP's capabilities. If the NRP does not respond, the primary attempts to announce a new NRP, provided there are available alternatives, using the `new_NRP` message server (L1, line 56). Once a valid NRP is confirmed, the primary DCN starts sending heartbeats on both networks. If no NRP is available, the primary transitions to the `FAILED` mode (implemented in the `ping_timed_out`, L1, line 24).

In the `BACKUP` mode, the DCN monitors heartbeats from the primary. To minimize false positives caused by temporary network disturbances, the heartbeat period and tolerance limits (i.e., the number of missed heartbeats allowed before declaring a timeout) must be carefully set. Since DCN redundancy typically relies on two independent network paths, the backup expects to receive a

```
msgsrv runMe() {
 switch(mode) {
  case 0: //WAITING :
   if (id == primary) {
    mode = PRIMARY;
    NRP_network++;
    if (NRP_network < NumberOfNetworks) {
     NRP_switch_id = NRPCandidates[NRP_network];
     if (NRP_network == 0) out1.new_NRP(id, NRP_network, NRP_switch_id);
     else out2.new_NRP(id, NRP_network, NRP_switch_id);
    } else NRP_network = NumberOfNetworks;
   } else mode = BACKUP;
   break;
  case 1: //PRIMARY :
   if (NRP_network == 0) {
    ping_pending = true;
    out1.pingNRP(id, NRP_switch_id) after(networkDelay);
    ping_timed_out() after(ping_timeout);
   } else {
    ping_pending = true;
    out2.pingNRP(id, NRP_switch_id) after(networkDelay);
    ping_timed_out() after(ping_timeout);
   }
   NRP_pending = true;
   break;
  case 2: //BACKUP :
   heartbeats_missed_1++;
   heartbeats_missed_2++;
   if (heartbeats_missed_1 > max_missed_heartbeats && heartbeats_missed_2 >
   ↪  max_missed_heartbeats) {
    if (heartbeats_missed_1==heartbeats_missed_2) {
     mode = PRIMARY;
     primary = id;
    } else {
     become_primary_on_ping_response = true;
     ping_pending = true;
     if (NRP_network == 0){
      out1.pingNRP(id, NRP_switch_id) after(networkDelay);
     } else {
      out2.pingNRP(id, NRP_switch_id) after(networkDelay);
     }
     ping_timed_out() after(ping_timeout);
    }
   } else if (heartbeats_missed_1 > max_missed_heartbeats || heartbeats_missed_2 >
   ↪  max_missed_heartbeats) {
    if (NRP_network==0 && heartbeats_missed_1 > max_missed_heartbeats) {
     ping_pending = true;
     out1.pingNRP(id, NRP_switch_id) after(networkDelay);
     ping_timed_out() after(ping_timeout);
    } else if (NRP_network ==1  && heartbeats_missed_2 > max_missed_heartbeats) {
     ping_pending = true;
     out2.pingNRP(id, NRP_switch_id) after(networkDelay);
     ping_timed_out() after(ping_timeout);
    }
    heartbeats_missed_1 = (heartbeats_missed_1>max_missed_heartbeats+1)?
    ↪  max_missed_heartbeats+1:heartbeats_missed_1;
    heartbeats_missed_2 = (heartbeats_missed_2>max_missed_heartbeats+1)?
    ↪  max_missed_heartbeats+1:heartbeats_missed_2;
   }
   break;
  case 3: //FAILED :
   break;
 }
 self.runMe() after(heartbeat_period);
}
```

Listing 2: (L2) Different modes of a DCN and their corresponding behavior in the message server `runMe`.

heartbeat on each path within every period. If heartbeats are missed on both paths simultaneously, it likely indicates a primary failure rather than a network issue. To optimize failover, NRP FD allows the backup to transition directly to `PRIMARY` mode when it detects simultaneous heartbeat timeouts, skipping the `pingNRP` exchange (L2, line 29-32). However, this optimization slightly increases the risk of dual primaries, which can be caught by model checking. We set the `max_missed_heartbeat` to 2 (L1, line 2). The variables `heartbeats_missed_1` and `heartbeats_missed_2` act as counters for tracking missed heartbeats on the two networks. These counters increase at each period and reset when a heartbeat is successfully received (in message server heartBeat, L1, line 37).

The backup DCN continuously monitors the count of missed heartbeats for each network. If both counters exceed the limit defined by `max_missed_heartbeat` (L2, line 29), the backup assumes a failure has occurred. We sets the variable `become_primary_on_ping_response` to true, which will be checked later in message server `pingNRP_response`. The backup then sends a `pingNRP` message to the NRP to check its reachability (L2, line 34-43). If the NRP responds, confirming its accessibility, the DCN transitions from `BACKUP` to `PRIMARY` mode (`pingNRP_response` in L1, line 25). Otherwise, if one of the counters exceeds, it will check the NRP reachability based on the network on which it is located. Finally, the `heartbeats_missed1` and `heartbeats_missed2` are bound accordingly.

In the `FAILED` mode, NRP FD remains inactive until it receives a manual acknowledgment confirming that the underlying issues causing the failure have been resolved.

In the previous work [10], we performed model checking and removed the transition from backup mode to the primary in simultaneous network failures, as it could lead to dual primaries. Here, while consistency remains the priority, we also strive to maintain availability by keeping the model aligned with the specification. Therefore, we retain this improvement in this paper and analyze the minimum possible time interval between failures that can still be considered non-simultaneous. We will discuss the timing analysis in the following sections.

Accuracy of the model. Our model closely reflects real-world conditions by accurately representing the network topology and DCN interactions. The decision to tolerate up to two lost heartbeats is based on the low bit error rate of Gigabit Ethernet and the fact that a heartbeat message can fit within a standard 1500-byte Ethernet frame. This indicates a low probability of losing heartbeat messages, particularly across two separate networks, thereby reducing the chance of false positives caused by regular disruptions. The reaction time, which is the time from a primary failure to when the backup takes over the primary role, is determined by the `heartbeat_period` and `max_missed_heartbeats`.

In [10], we use a rationale to set the timing parameters. In this paper, we changed the numbers to reduce the size of the state space during model checking while keeping the numbers proportional to real-world settings. Here, we define the `heartbeat_period` as 20 time units and set the `ping_timeout` and `nrp_timeout` to 10 time units. The `networkDelay` is considered as 1 unit of time.

When DCNs ping NRP, we use the keyword `after` and set it to `networkDelay`. These values are chosen to be proportional to real-world values while preserving the sequence of message exchanges. Although they may vary slightly, our model should handle all timing events within one period of 20 time units. We use the `after` construct, where the execution order needs to be respected.

5 Time Analysis of NRP FD: Consistency, Availability, Latency

For the structure in Figure 1, we claim that the code in the previous section will not lead to dual primaries unless we get "simultaneous" failures in the two networks. In the code in Listing 2, on line 30, we see the comparison between the number of missed heartbeats in the two networks. If the number of missed heartbeats is the same for both networks then the backup takes over and becomes the primary. The reasoning is that the "simultaneous" failure of both networks is very unlikely, so the algorithm assumes that the primary is down in this situation. This relaxes the requirement for consistency by allowing dual primaries in rare cases, thereby improving availability, meaning that the backup can successfully take over. Dual primaries can only occur when network failures are "simultaneous," something that can be made arbitrarily unlikely.

Here, we quantify "simultaneous" and show how to construct a Rebeca program that nondeterministically allows all possible non-simultaneous switch failures. We then perform model checking on the resulting program and verify that dual primaries cannot happen when the time between switch failures is sufficiently long.

Let H be the heartbeat period. The primary node will send the n^{th} heartbeat at time $nH + P$, where P is the ping timeout. The addition of P is so that the primary verifies that it still has access to the NRP before sending a heartbeat. The program is designed to use P as a fixed offset. It pings the NRP at time nH and then waits for P time units. If it receives a response to the ping within this time period, then it will send a heartbeat at time $nH + P$.

Assume that the delay on each network hop is N time units. In Figure 1, there are three switches and, therefore, four network hops from the primary to the backup node. The backup will receive the heartbeat if the first switch does not fail before $nH + P + N$, the second switch does not fail before $nH + P + 2N$, and the third switch does not fail before $nH + P + 3N$.

Let t_A be the time at which network A fails for the first time, and t_B the time at which network B fails for the first time. We define the network failures to be "simultaneous" if t_A and t_B satisfy the following inequalities for some natural number n:

$$\begin{aligned} (n-1)H + P + iN < t_A \leq nH + P + iN \\ (n-1)H + P + jN < t_B \leq nH + P + jN \end{aligned} \tag{1}$$

where $i, j \in \{1, 2, 3\}$ denotes which switch failed (first, second, or third along the path from the primary to the backup).

Assume without loss of generality that $t_A \leq t_B$. For such a simultaneous failure, the earliest possible t_A occurs if the first switch fails ($i = 1$) and

$$t_A = (n-1)H + P + N + 1 \tag{2}$$

The +1 accounts for strict inequality (time is quantized here).

In such a simultaneous failure, the latest possible time t_B for network B to fail is

$$t_B = nH + P + 3N \tag{3}$$

where we have chosen $j = 3$. Hence, the largest possible time difference between failures for them to be considered "simultaneous" is

$$t_B - t_A = H + 2N - 1. \tag{4}$$

Hence, our claim is that our controllers will not yield dual primaries as long as

$$|t_B - t_A| \geq H + 2N \ . \tag{5}$$

To verify this using model checking, we augment the Rebeca program with a `FailureController` reactive class, shown in Listing 3 (L3). This code assumes a heartbeat period of $H = 20$ and a network latency $N = 1$. The constructor, on line 8, ensures that the first failure does not occur in the first two heartbeat periods. After two heartbeat periods, the handler `chooseFailureTime` is invoked. It nondeterministically chooses a possible failure time between zero and 19 and then invokes `maybeFail` after that time (L3, line 18). The `maybeFail` handler first decides nondeterministically whether a failure will occur at this time, and, if so, nondeterministically selects one of the six switches to fail. If no failure occurs, then (on line 34) it repeats `maybeFail` one time unit later. If a failure does occur, then (on line 32), it ensures that `min_interval` elapses before the next possible failure. We set `min_interval` to 23 (L3, line 4), based on the inequality (5), considering that the heartbeat is 20 and the N is 1 unit of time, and +1 for strict inequality (to be sure that we will not have a simultaneous failure).

We implemented the algorithm in Rebeca and provided an assertion that inequality (5) holds. On our first attempt, the assertion failed. Afra's counterexample revealed a corner case in our code in which the behavior did not match the assumptions stated above. Specifically, the Primary sends a ping to the NRP and waits for a timeout period. If it receives the response to the ping within that timeout period, then it sends a heartbeat. If not, it sends a message to acquire a new NRP. In this latter event, our first implementation did not send a heartbeat message in this cycle. The next heartbeat message was sent one full cycle later.

We now face a choice. We can either refine the assumptions in the derivation above and derive a new inequality,

$$|t_B - t_A| \geq 2H + 2N \ , \tag{6}$$

or we can modify the code to match the original assumption. Either strategy works. The model checker verifies that our first implementation satisfies (6) and that a slightly modified implementation satisfies (5). This shows how the details of the implementation can affect the timings in a way that is hard to notice by analytical approaches.

```
env int heartbeat_period = 20;
env int networkDelay = 1;
...
env int min_interval = 23;
...
reactiveclass FailureController(10) {
    knownrebecs {
        Switch switchA1, switchA2, switchA3, switchB1, switchB2, switchB3;
    }
    statevars {}
    FailureController () {
        self.chooseFailureTime() after(2*heartbeat_period);  // Let startup phase
        ↪  pass.
    }
    msgsrv chooseFailureTime() {
        int failure_time = ? (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
        ↪  16, 17, 18, 19);
        self.maybeFail() after (failure_time);
    }
    msgsrv maybeFail() {
        boolean failure = ? (false, true);
        byte s = ? (1, 2, 3, 4, 5, 6);
        if (failure) {
            if (s == 1) {
                switchA1.switchFail();
            } else if (s == 2) {
                switchA2.switchFail();
            } else if (s == 3) {
                switchA3.switchFail();
            } else if (s == 4) {
                switchB1.switchFail();
            } else if (s == 5) {
                switchB2.switchFail();
            } else if (s == 6) {
                switchB3.switchFail();
            }
            maybeFail() after (min_interval);
        } else {
            maybeFail() after (1);
        }
    }
}
....
main {
    @Priority(1) FailureController failureC(switchA1, switchA2, switchA3, switchB1,
    ↪  switchB2, switchB3):();

    @Priority(1) Switch switchA1():(1, 0, true , switchA2 , switchA2 ,
    ↪  switchA1failtime, DCN1);
    ...
}
```

Listing 3: FailureController reactive class.

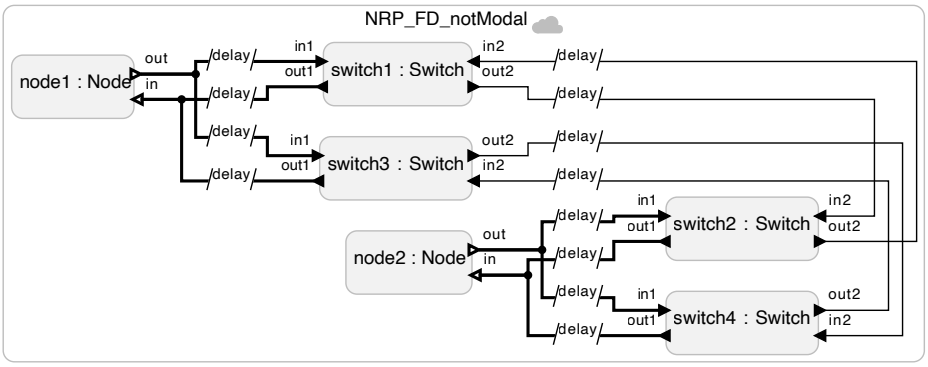

Fig. 3: Diagram of a Lingua Franca realization of a two-node/two-network example (closed).

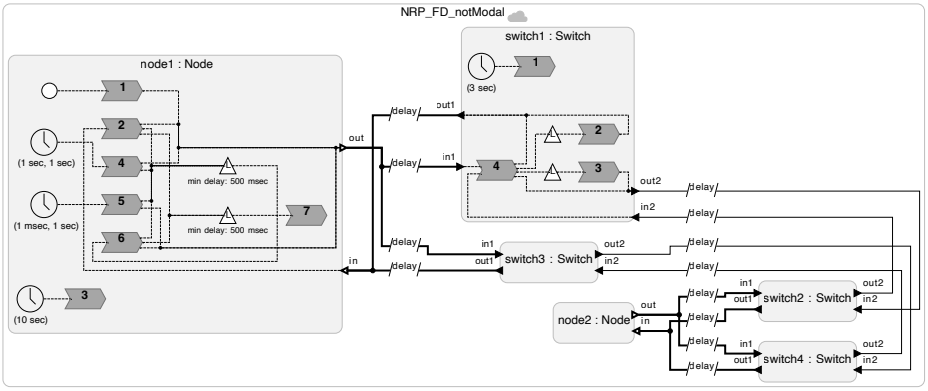

Fig. 4: Diagram of a Lingua Franca realization of a two-node/two-network example where node 1 and switch1 are expanded.

6 Implementation and Simulation of NRP FD using Lingua Franca

Lingua Franca bridges the modeling in Timed Rebeca to the implementation, most importantly by aligning the logical and physical time. An (automatically generated) diagram of an LF realization of a two-node/two-network version of the NRP system is shown in Figure 3. One of the two nodes is expanded in Figure 4, showing some implementation details. The realization has the form of a modal model [21], where a finite-state machine (represented by bubbles and arcs in the diagram) governs the switching between modes of operation, and each mode defines reactions (represented by grey chevrons in the diagram). The reactions encapsulate C code that reacts to timestamped input events.

The LF tools translate the node into a C program that can be installed in a deployed system, and, therefore, the code used for modeling and simulation

is the same as the code used in deployment. Moreover, the generated C code for the nodes can be run on the same hardware it will be deployed on, thereby enabling hardware-in-the-loop simulation. The only difference is that in modeling and simulation, the simulated network switches determine the timestamps of messages that arrive at a node, whereas in deployment, a local clock measuring physical time will determine the timestamps. Consequently, the simulation model can reproduce test cases that may be seen in the field, and when those cases are seen in the field, the behavior will match that of the simulation.

The figure shows a test case that simulates particular network delays and specified times at which components fail. For example, the connections between the nodes and the switches are decorated with *after* delays, notated `/delay/`, where `delay` is a parameter specifying a time value. These delay annotations determine the timestamps at which the nodes receive messages in the simulation, whereas in a deployed system, these timestamps will be determined by a physical clock.

7 Discussion

The interaction of concurrency and timing makes it difficult to design correct algorithms for distributed timed systems and predict their behavior entirely. Using timed actors, we can faithfully model the distributed event-driven algorithms and programs, and model checking helps reveal the possible problems or bugs in the algorithm and the program. Using model checking where the models are more abstract comparing to what presented here would not necessarily reveal how the implementation details affect the behavior. We showed the method in this paper and our experiment demonstrated the possible complications of interaction of concurrency and timing.

The method is general and can be applied to other event-driven reactive systems. For example, the same approach can be used in autonomous driving (AD) or communication protocols. In both cases, components can be modeled as actors interacting with each other. The behavior, along with the fault injection patterns, can be easily applied at different detail levels. For instance, in an AD system, the detail could be coarse by describing the behavior of interacting driving functions or more fine by modeling the actual communication traffic running between two (or more) electronic control units implementing these functions.

Timed Rebeca is used for timing analysis in different applications, including analysis of timing properties in interoperable medical systems by Zarneshan et al. [31], and analyzing real-time wireless sensor and actuator networks by Khamespanah et al. [13] where Gul Agha is a co-author. In [12], Khamespanah et al. compared analytical methods and model checking using Afra for schedulability analysis of WSAN applications. This paper is different in its approach; here we model failures and implementation details, and reveal how these details can change the timing features of the system.

A Rebeca Syntax

The abstract syntax of Timed Rebeca from [24] is shown in Figure 5.

$$
\begin{aligned}
Model &::= Class^{*}\ Main \\
Main &::= \textbf{main}\ \{\ InstanceDcl^{*}\ \} \\
InstanceDcl &::= className\ rebecName(\langle rebecName\rangle^{*}) : (\langle literal\rangle^{*}); \\
Class &::= \textbf{reactiveclass}\ className\ \{\ KnownRebecs\ Vars\ MsgSrv^{*}\ \} \\
KnownRebecs &::= \textbf{knownrebecs}\ \{\ VarDcl^{*}\ \} \\
Vars &::= \textbf{statevars}\ \{\ VarDcl^{*}\ \} \\
VarDcl &::= type\ \langle v\rangle^{+}; \\
MsgSrv &::= \textbf{msgsrv}\ methodName(\langle type\ v\rangle^{*})\ \{\ Stmt^{*}\ \} \\
Stmt &::= v = e;\ |\ v =?(e, \langle e\rangle^{+});\ |\ Call;\ |\ \textbf{delay}(t);\ |\ if\ (e)\ \{\ Stmt^{*}\ \}[else\ \{\ Stmt^{*}\ \}] \\
Call &::= rebecName.methodName(\langle e\rangle^{*})\ [\textbf{after}(t)]\ [\textbf{deadline}(t)]
\end{aligned}
$$

Fig. 5: An abstract syntax for Timed Rebeca. The identifiers, className, rebecName, methodName, literal, and type, are self-explanatory. The identifier v denotes a variable. The symbol e denotes an expression, which can be either arithmetic, boolean, or a non-deterministic choice. Angular brackets $\langle ... \rangle$ serve as meta-parenthesis, with superscript + denoting at least one repetition and superscript $*$ denoting zero or more repetitions. Meanwhile, using $\langle ... \rangle$ with repetition indicates a comma-separated list. Square brackets [...] indicate that the enclosed text is optional [24].

B Timed Rebeca model of the NRP FD

```
// Timed Rebeca model for NRP FD.
// This program models a redundant fault tolerant system where a primary node, if and when it fails,
// is replaced by a backup node. This version models network switch failures that are spaced
// sufficiently far apart in time that dual primaries do not emerge.  To manage the state-space
// size, the heartbeat period and other time values are reduced.
//
// The protocol is described in this paper:
// Bjarne Johansson; Mats Rgberger; Alessandro V. Papadopoulos; Thomas Nolte, "Consistency Before
// Availability: Network Reference Point based Failure Detection for Controller Redundancy," paper
// draft 8/15/23.
//
// The key idea in this protocol is that when a backup fails to detect the heartbeats of a primary
// node, it becomes primary only if it has access to Network Reference Point (NRP), which is a point
// in the network.
//
// The Primary sends heartbeats on two networks,
// if the Backup receives the heartbeats from both networks then all is fine.
// If it receives the heartbeat only from one network the Backup pings the NRP, if NRP replies all is fine,
// if not
// If Backup misses heartbeats on both networks then it assumes that the Primary failed and pings NRP,
// if NRP replies, Backup becomes the Primary
// if not ...
//
// The Rebeca code is adopted from the LF code by Edward Lee and Marjan Sirjani, and refined by Bahman Pourvatan

// Timing configuration
env int heartbeat_period = 20;
env int max_missed_heartbeats = 2;
env int ping_timeout = 10;
env int nrp_timeout = 10; // Timeout for requesting a new NRP.
// Node Modes
env byte WAITING = 0;
env byte PRIMARY = 1;
env byte BACKUP = 2;
env byte FAILED = 3;
env byte NumberOfNetworks = 2;
// for testing
env int fails_at_time = 0; //zero for no failure
env int switchA1failtime = 0;
env int switchA2failtime = 0;
env int switchA3failtime = 0;
env int switchB1failtime = 0;
env int switchB2failtime = 0;
env int switchB3failtime = 0;
env int node1failtime = 0;
env int node2failtime = 0;
env int networkDelay = 1;

// Minimum time between switch failures is H + 2N + 1, where H is the heartbeat_period and N is the networkDelay.
// The +1 is needed because of the nondeterministic ordering of message handlers.
// If a switch receives a switchFail() message at some time t, it nondeterministically may forward
// a heartbeat message that happens to arrive at the same time.
// The +1 ensures that the failure occurs after the message has been forwarded.
env int min_interval = 23;

// Generate switchFailure messages to switches that are nondeterministically chosen.
// This class ensures a minimum interval of min_interval between any two switch failures.
reactiveclass FailureController(10) {
    knownrebecs {
        Switch switchA1, switchA2, switchA3, switchB1, switchB2, switchB3;
    }
    statevars {}
    FailureController () {
        self.chooseFailureTime() after(2*heartbeat_period);  // Let startup phase pass.
    }
    msgsrv chooseFailureTime() {
        int failure_time = ? (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19);
        self.maybeFail() after (failure_time);
    }
    msgsrv maybeFail() {
        boolean failure = ? (false, true);
        byte s = ? (1, 2, 3, 4, 5, 6);
        if (failure) {
            if (s == 1) {
                switchA1.switchFail();
            } else if (s == 2) {
                switchA2.switchFail();
            } else if (s == 3) {
                switchA3.switchFail();
```

```
            } else if (s == 4) {
                switchB1.switchFail();
            } else if (s == 5) {
                switchB2.switchFail();
            } else if (s == 6) {
                switchB3.switchFail();
            }
            maybeFail() after (min_interval);
        } else {
            maybeFail() after (1);
        }
    }
}

reactiveclass Node (20){
    knownrebecs {
        Switch out1, out2;
    }
    statevars {
           byte mode;
           int id;
           int [2] NRPCandidates;
           int heartbeats_missed_1;
           int heartbeats_missed_2;
           int NRP_network;
           int NRP_switch_id;
           boolean NRP_pending;
           boolean become_primary_on_ping_response;
           int primary;
           boolean ping_pending;
    }
    Node (int Myid, int Myprimary, int NRPCan1_id, int NRPCan2_id, int myFailTime) {
        id = Myid;
        NRPCandidates[0] = NRPCan1_id;
        NRPCandidates[1] = NRPCan2_id;
        heartbeats_missed_1 = 0;
        heartbeats_missed_2 = 0;
        NRP_network = -1;
        NRP_switch_id = -1;
        NRP_pending = true;
        become_primary_on_ping_response = false;
        primary = Myprimary;
        ping_pending = false;
        mode = WAITING;
        if(myFailTime!=0) nodeFail() after(myFailTime);
        runMe();
    }
    // Check whether a response was received to request_new_NRP() and fail if not.
    msgsrv new_NRP_request_timed_out() {
        if (mode == BACKUP) {
            if (NRP_pending) {
                NRP_pending = false;
                mode = FAILED;
            }
        }
    }
    // logical action ping_timed_out(ping_timeout)
    msgsrv ping_timed_out() {
        if (mode == BACKUP) {
            if (ping_pending) {
                // Backup node did not get a response to the ping of the NRP.
                ping_pending = false;
                NRP_network++;
                // Request a new NRP, unless it pinged the NRP because both networks had missed heartbeats.
                if (become_primary_on_ping_response) {
                    become_primary_on_ping_response = false;
                    mode = FAILED;
                } else if (NRP_network < NumberOfNetworks) {
                    // Request a new NRP should be possible.
                    NRP_pending = true;
                    NRP_switch_id = NRPCandidates[NRP_network];
                    if (NRP_network == 0) out1.request_new_NRP(id) after(networkDelay);
                    else out2.request_new_NRP(id) after(networkDelay);
                    self.new_NRP_request_timed_out() after(nrp_timeout);
                } else {
                    NRP_network = NumberOfNetworks;
                    mode = FAILED; // Operator intervention required.
                }
            }
        } else if (mode == PRIMARY) {
            if (ping_pending) {
                // NRP did not respond to ping. Try to find a new NRP, if possible, and fail otherwise.
                NRP_pending = true;
                NRP_network++;
```

```
164                 if (NRP_network < NumberOfNetworks) {
165                     // Select the NRP candidate on the new network and ping it.
166                     NRP_switch_id = NRPCandidates[NRP_network];
167                     if (NRP_network == 0) out1.new_NRP(id, NRP_network,NRP_switch_id)
                        ↪  after(networkDelay);
168                     else out2.new_NRP(id, NRP_network,NRP_switch_id) after(networkDelay);
169                 } else {
170                     NRP_network = NumberOfNetworks;
171                     mode = FAILED; // Operator intervention required.
172                 }
173             } else {
174                out1.heartBeat(0, id) after(networkDelay);
175                out2.heartBeat(1, id) after(networkDelay);
176             }
177         }
178     }
179     msgsrv pingNRP_response(int mid) {
180         if (mode == BACKUP) {
181             ping_pending = false;
182             if (become_primary_on_ping_response) {
183                 // Confirmed that NRP is accessible even though heartbeats are missing on both
                    ↪  networks.
184                 become_primary_on_ping_response = false;
185                 mode = PRIMARY;
186                 primary = id;
187             }
188         } else if (mode == PRIMARY) {
189             ping_pending = false;
190             if (NRP_pending) {
191                 NRP_pending = false; // NRP is confirmed.
192             }
193         }
194     }
195     msgsrv new_NRP(int mid, int mNRP_network, int mNRP_switch_id) {
196         NRP_network = mNRP_network;
197         NRP_switch_id = mNRP_switch_id;
198         if (NRP_network == 0) heartbeats_missed_1 = 0;
199         else heartbeats_missed_2 = 0;
200 
201         NRP_pending = false;
202     }
203     msgsrv request_new_NRP(int origin) {
204         NRP_network++;
205         if (NRP_network < NumberOfNetworks) {
206             NRP_switch_id = NRPCandidates[NRP_network];
207             if (NRP_network == 0) out1.new_NRP(id, NRP_network, NRP_switch_id);
208             else out2.new_NRP(id, NRP_network, NRP_switch_id);
209         }
210     }
211     msgsrv runMe() {
212         switch(mode) {
213             case 0: //WAITING :
214                 if (id == primary) {
215                     mode = PRIMARY;
216                     NRP_network++;
217                     if (NRP_network < NumberOfNetworks) {
218                         NRP_switch_id = NRPCandidates[NRP_network];
219                         if (NRP_network == 0) out1.new_NRP(id, NRP_network, NRP_switch_id);
220                         else out2.new_NRP(id, NRP_network, NRP_switch_id);
221                      } else NRP_network = NumberOfNetworks;
222                  } else mode = BACKUP;
223                  break;
224              case 1: //PRIMARY :
225                  if (NRP_network == 0) {
226                      ping_pending = true;
227                      out1.pingNRP(id, NRP_switch_id) after(networkDelay);
228                      ping_timed_out() after(ping_timeout);
229                  } else {
230                      ping_pending = true;
231                      out2.pingNRP(id, NRP_switch_id) after(networkDelay);
232                      ping_timed_out() after(ping_timeout);
233                  }
234                  NRP_pending = true;
235                  break;
236              case 2: //BACKUP :
237                  heartbeats_missed_1++;
238                  heartbeats_missed_2++;
239                  if (heartbeats_missed_1 > max_missed_heartbeats && heartbeats_missed_2 >
                     ↪  max_missed_heartbeats) {
240                      if (heartbeats_missed_1==heartbeats_missed_2) { // Simultaneous
                         ↪  heartbeat misses.
241                          mode = PRIMARY;
242                          primary = id;
243                      } else {
```

```
                        become_primary_on_ping_response = true;
                        ping_pending = true;
                         if (NRP_network == 0){
                             out1.pingNRP(id, NRP_switch_id) after(networkDelay);
                         } else {
                             out2.pingNRP(id, NRP_switch_id) after(networkDelay);
                         }
                         ping_timed_out() after(ping_timeout);
                    }
                } else if (heartbeats_missed_1 > max_missed_heartbeats ||
                ↪  heartbeats_missed_2 > max_missed_heartbeats) {
                    // Heartbeat missed on one network but not yet on the other.
                    // Ping the NRP to make sure we retain access to it so that we can be an
                    ↪  effective backup.
                     if (NRP_network==0 && heartbeats_missed_1 > max_missed_heartbeats) {
                        ping_pending = true;
                        out1.pingNRP(id, NRP_switch_id) after(networkDelay);
                        ping_timed_out() after(ping_timeout);
                    } else if (NRP_network ==1  && heartbeats_missed_2 >
                    ↪  max_missed_heartbeats) {
                        ping_pending = true;
                        out2.pingNRP(id, NRP_switch_id) after(networkDelay);
                        ping_timed_out() after(ping_timeout);
                    }
                    heartbeats_missed_1 = (heartbeats_missed_1>max_missed_heartbeats+1)?
                    ↪  max_missed_heartbeats+1:heartbeats_missed_1;
                        heartbeats_missed_2 = (heartbeats_missed_2>max_missed_heartbeats+1)?
                        ↪  max_missed_heartbeats+1:heartbeats_missed_2;
                }
                break;
             case 3: //FAILED :
                break;
        }
        self.runMe() after(heartbeat_period);
    }
    msgsrv heartBeat(byte networkId, int senderid) {
        if (mode==BACKUP) {
            if (networkId == 0) heartbeats_missed_1 = 0;
            else heartbeats_missed_2 = 0;
        }
    }
    msgsrv nodeFail() {
        primary=-1;
        mode = FAILED;
        NRP_network=-1;
        NRP_switch_id=-1;
        heartbeats_missed_1 = 0;
        heartbeats_missed_2 = 0;
        NRP_pending = true;
        become_primary_on_ping_response = false;
        ping_pending = false;
    }
}

reactiveclass Switch(10) {
    knownrebecs {
    }
    statevars {
        byte mynetworkId;
        int id;
        boolean failed;
        boolean amINRP;
        boolean terminal;
        Node nodeTarget1;
        Switch switchTarget2;
        Switch switchTarget1;
    }
    Switch (int myid, byte networkId, boolean term, Switch s1, Switch s2, int myFailTime,
    ↪  Node n1) {
        mynetworkId = networkId;
        id = myid;
        amINRP = false;
        failed = false;
        switchTarget1 = s1;
        switchTarget2 = s2;
        terminal=term;
        nodeTarget1=n1;
        if (myFailTime!=0) switchFail() after(myFailTime);
    }
    msgsrv switchFail() {
        failed = true;
        amINRP=false;
    }
```

```
321     msgsrv pingNRP_response(int senderNode) {
322         if (!failed) {
323             if (terminal && senderNode <= 100) {
324                 nodeTarget1.pingNRP_response(id) after(networkDelay); //Pass back
325             } else if (senderNode>id) {
326                 switchTarget1.pingNRP_response(id) after(networkDelay);
327             } else {
328                 switchTarget2.pingNRP_response(id) after(networkDelay);
329             }
330         }
331     }
332     msgsrv request_new_NRP(int senderNode) {
333         if (!failed) {
334             if (terminal && senderNode < 100) nodeTarget1.request_new_NRP(id)
                ↪  after(networkDelay);
335             else if (senderNode > id) switchTarget1.request_new_NRP(id) after(networkDelay);
336             else switchTarget2.request_new_NRP(id) after(networkDelay);
337         }
338     }
339     msgsrv pingNRP(int senderNode, int NRP) {
340         if (!failed) {
341             if (NRP == id) {
342                 if (senderNode < 100) switchTarget2.pingNRP_response(id) after(networkDelay);
                    ↪  //Response
343                 else nodeTarget1.pingNRP_response(id) after(networkDelay);
344             } else {
345                 if(senderNode < 100) {
346                     if(senderNode>id) switchTarget1.pingNRP(id, NRP) after(networkDelay);
347                     else switchTarget2.pingNRP(id, NRP) after(networkDelay);
348                 } else {
349                     switchTarget2.pingNRP(id, NRP) after(networkDelay);
350                 }
351             }
352         }
353     }
354     msgsrv new_NRP(int senderNode, int mNRP_network, int mNRP_switch_id) {
355         if (!failed) {
356             if (id == mNRP_switch_id) amINRP=true;
357             else amINRP=false;
358 
359             if (terminal && senderNode < 100) nodeTarget1.new_NRP(id, mNRP_network,
                ↪  mNRP_switch_id); //Pass back
360             else if(senderNode>id) switchTarget1.new_NRP(id, mNRP_network, mNRP_switch_id);
361             else switchTarget2.new_NRP(id, mNRP_network, mNRP_switch_id);
362         }
363     }
364     msgsrv heartBeat(byte networkId, int senderNode) {
365         if (!failed) {
366             if (terminal && senderNode < 100) nodeTarget1.heartBeat(networkId,id)
                ↪  after(networkDelay);
367             else if (senderNode > id) switchTarget1.heartBeat(networkId,id)
                ↪  after(networkDelay);
368             else switchTarget2.heartBeat(networkId,id) after(networkDelay);
369         }
370     }
371 }
372 main {
373     @Priority(1) FailureController failureC(switchA1, switchA2, switchA3, switchB1, switchB2,
    ↪  switchB3):();
374 
375     @Priority(1) Switch switchA1():(1, 0, true , switchA2 , switchA2 , switchA1failtime,
    ↪  DCN1);
376     @Priority(1) Switch switchA2():(2 ,0, false , switchA1 , switchA3 , switchA2failtime,
    ↪  null);
377     @Priority(1) Switch switchA3():(3, 0, true , switchA2 , switchA2 , switchA3failtime,
    ↪  DCN2);
378     @Priority(1) Switch switchB1():(4, 1, true , switchB2 , switchB2 , switchB1failtime,
    ↪  DCN1);
379     @Priority(1) Switch switchB2():(5, 1, false , switchB1 , switchB3 , switchB2failtime,
    ↪  null);
380     @Priority(1) Switch switchB3():(6, 1, true , switchB2 , switchB2 , switchB3failtime,
    ↪  DCN2);
381 
382     @Priority(2) Node DCN1(switchA1, switchB1):(100, 100, 1, 4, node1failtime);
383     @Priority(2) Node DCN2(switchA3, switchB3):(101, 100, 3, 6, node2failtime);
384 }
```

References

1. Agha, G.: An overview of actor languages. In: Wegner, P., Shriver, B.D. (eds.) Proceedings of the 1986 SIGPLAN Workshop on Object-Oriented Programming, OOPWORK 1986, Yorktown Heights, New York, USA, June 9-13, 1986. pp. 58–67. ACM (1986). https://doi.org/10.1145/323779.323743, https://doi.org/10.1145/323779.323743
2. Agha, G., Hewitt, C.: Actors: A conceptual foundation for concurrent object-oriented programming. In: Shriver, B.D., Wegner, P. (eds.) Research Directions in Object-Oriented Programming, pp. 49–74. MIT Press (1987)
3. Agha, G.A.: Actors: a Model of Concurrent Computation in Distributed Systems (Parallel Processing, Semantics, Open, Programming Languages, Artificial Intelligence). Ph.D. thesis, University of Michigan, USA (1985), http://hdl.handle.net/2027.42/160629
4. Åkerberg, J., Furunäs Åkesson, J., Gade, J., Vahabi, M., Björkman, M., Lavassani, M., Nandkumar Gore, R., Lindh, T., Jiang, X.: Future industrial networks in process automation: Goals, challenges, and future directions. Applied Sciences **11**(8), 3345 (2021)
5. Atkinson, R.R., Hewitt, C.: Parallelism and synchronization in actor systems. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977. pp. 267–280. ACM (1977). https://doi.org/10.1145/512950.512975, https://doi.org/10.1145/512950.512975
6. Brewer, E.A.: Towards robust distributed systems (abstract). In: Neiger, G. (ed.) Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing, July 16-19, 2000, Portland, Oregon, USA. p. 7. ACM (2000). https://doi.org/10.1145/343477.343502, https://doi.org/10.1145/343477.343502
7. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: Proceedings of the 3rd international joint conference on Artificial intelligence. pp. 235–245. Morgan Kaufmann Publishers Inc. (1973), http://ijcai.org/Proceedings/73/Papers/027B.pdf
8. Hewitt, C., Bishop, P.B., Greif, I., Smith, B.C., Matson, T., Steiger, R.: Actor induction and meta-evaluation. In: Fischer, P.C., Ullman, J.D. (eds.) Conference Record of the ACM Symposium on Principles of Programming Languages, Boston, Massachusetts, USA, October 1973. pp. 153–168. ACM Press (1973). https://doi.org/10.1145/512927.512942, https://doi.org/10.1145/512927.512942
9. Johansson, B., Lee, E., Sirjani, M., Moezkarimi, Z., Pourvatan, B., Marksteiner, S., Papadopoulos, A.: Systematic test case generation for distributed redundant controllers using model checking (extended abstract) (2024)
10. Johansson, B., Pourvatan, B., Moezkarimi, Z., Papadopoulos, A., Sirjani, M.: Formal verification of consistency for systems with redundant controllers. Electronic Proceedings in Theoretical Computer Science **399**, 169–191 (2024)
11. Johansson, B., Rågberger, M., Papadopoulos, A., Nolte, T.: Consistency before availability: Network reference point based failure detection for controller redundancy. In: ETFA. pp. 1–8 (2023)
12. Khamespanah, E., Mohaqeqi, M., Ashjaei, M., Sirjani, M.: Schedulability analysis of WSAN applications: Outperformance of a model checking approach. In: 27th IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2022, Stuttgart, Germany, September 6-9, 2022. pp. 1–8. IEEE (2022). https://doi.org/10.1109/ETFA52439.2022.9921644, https://doi.org/10.1109/ETFA52439.2022.9921644

13. Khamespanah, E., Sirjani, M., Mechitov, K., Agha, G.: Modeling and analyzing real-time wireless sensor and actuator networks using actors and model checking. Int. J. Softw. Tools Technol. Transf. **20**(5), 547–561 (2018). https://doi.org/10.1007/S10009-017-0480-3, https://doi.org/10.1007/s10009-017-0480-3
14. Khamespanah, E., Sirjani, M., Sabahi-Kaviani, Z., Khosravi, R., Izadi, M.: Timed rebeca schedulability and deadlock freedom analysis using bounded floating time transition system. Science of Computer Programming **98**, 184–204 (2015). https://doi.org/10.1016/j.scico.2014.07.005
15. Khosravi, R., Khamespanah, E., Ghassemi, F., Sirjani, M.: Actors upgraded for variability, adaptability, and determinism. In: Workshop on State-of-the-Art of Active Objects. pp. 226–260 (2024). https://doi.org/10.1007/978-3-031-51060-1˙9
16. Leander, B., Johansson, B., Lindström, T., Holmgren, O., Nolte, T., Papadopoulos, A.V.: Dependability and security aspects of network-centric control. In: 2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA). pp. 1–8. IEEE (2023). https://doi.org/10.1109/ETFA54631.2023.10275344
17. Lee, E.A., Akella, R., Bateni, S., Lin, S., Lohstroh, M., Menard, C.: Consistency vs. availability in distributed cyber-physical systems. ACM Trans. Embed. Comput. Syst. **22**(5s), 138:1–138:24 (2023). https://doi.org/10.1145/3609119, https://doi.org/10.1145/3609119
18. Lohstroh, M., Lee, E.A.: Deterministic actors. In: Forum on Specification and Design Languages (FDL), (September 2-4 2019)
19. Lohstroh, M., Menard, C., Bateni, S., Lee, E.A.: Toward a lingua franca for deterministic concurrent systems. ACM Transactions on Embedded Computing Systems (TECS) **20**(4), Article 36 (2021)
20. Lohstroh, M., Schoeberl, M., Goens, A., Wasicek, A., Gill, C.D., Sirjani, M., Lee, E.A.: Actors revisited for time-critical systems. In: Proceedings of the 56th Annual Design Automation Conference 2019, DAC 2019, Las Vegas, NV, USA, June 02-06, 2019. p. 152. ACM (2019). https://doi.org/10.1145/3316781.3323469, https://doi.org/10.1145/3316781.3323469
21. Schulz-Rosengarten, A., Hanxleden, R.v., Lohstroh, M., Lee, E.A., Bateni, S.: Polyglot modal models through lingua franca. In: Cyber-Physical Systems and Internet of Things Week (CPS-IoT). pp. 337–242 (2023)
22. Simion, A., Bira, C.: A review of redundancy in plc-based systems. Advanced Topics in Optoelectronics, Microelectronics, and Nanotechnologies XI **12493**, 269–276 (2023). https://doi.org/10.1117/12.2644462
23. Sirjani, M.: Rebeca: Theory, applications, and tools. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.P. (eds.) Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures. Lecture Notes in Computer Science, vol. 4709, pp. 102–126. Springer (2006). https://doi.org/10.1007/978-3-540-74792-5“˙5
24. Sirjani, M.: Power is overrated, go for friendliness! expressiveness, faithfulness, and usability in modeling: The actor experience. In: Lohstroh, M., Derler, P., Sirjani, M. (eds.) Principles of Modeling - Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 10760, pp. 423–448. Springer (2018). https://doi.org/10.1007/978-3-319-95246-8“˙25
25. Sirjani, M., de Boer, F.S., Jaghoori, M.M.: Task scheduling in rebeca. In: NWPT 2007. pp. 16–18 (2007)

26. Sirjani, M., Khamespanah, E.: On time actors. In: Ábrahám, E., Bonsangue, M.M., Johnsen, E.B. (eds.) Theory and Practice of Formal Methods. Lecture Notes in Computer Science, vol. 9660, pp. 373–392. Springer (2016)
27. Sirjani, M., Lee, E.A., Khamespanah, E.: Model checking software in cyberphysical systems. In: COMPSAC 2020. pp. 1017–1026. IEEE (2020)
28. Sirjani, M., Lee, E.A., Khamespanah, E.: Verification of cyberphysical systems. Mathematics **8**(7) (2020). https://doi.org/10.3390/math8071068
29. Sirjani, M., Movaghar, A., Mousavi, M.: Compositional verification of an object-based model for reactive systems. In: AVoCS 2001 (2001), https://rebeca-lang.org/assets/papers/2001/CompositionalVerificationOfAnObject-BasedModelForReactiveSystems.pdf
30. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using rebeca. Fundam. Informaticae **63**(4), 385–410 (2004)
31. Zarneshan, M., Ghassemi, F., Khamespanah, E., Sirjani, M., Hatcliff, J.: Specification and verification of timing properties in interoperable medical systems. Log. Methods Comput. Sci. **18**(2) (2022). https://doi.org/10.46298/LMCS-18(2:13)2022, https://doi.org/10.46298/lmcs-18(2:13)2022