Formal Verification of Consistency for Systems with Redundant Controllers

Bjarne Johansson
ABB AB, Västerås, Sweden
Mälardalen University, Västerås, Sweden
bjarne.johansson@se.abb.com

Bahman Pourvatan Zahra Moezkarimi Alessandro Papadopoulos Marjan Sirjani
Mälardalen University, Västerås, Sweden
firstname.lastname@mdu.se

A potential problem that may arise in the domain of distributed control systems is the existence of more than one primary controller in redundancy plans that may lead to inconsistency. An algorithm called NRP FD is proposed to solve this issue by prioritizing consistency over availability. In this paper, we demonstrate how by using modeling and formal verification, we discovered an issue in NRP FD where we may have two primary controllers at the same time. We then provide a solution to mitigate the identified issue, thereby enhancing the robustness and reliability of such systems.

1 Introduction

Control systems are essential in the automation solution of domains such as offshore oil extraction, refineries, and hydropower plants - sectors where downtime can lead to significant financial losses or even life-threatening incidents. These automation solutions incorporate redundancy to mitigate the risk of unplanned downtime due to hardware failures by duplicating critical components like controllers. The common approach is standby redundancy, where an active primary controller manages the process, and a passive backup is ready to take over in case of primary failure [21]. These controllers, or Distributed Controller Nodes (DCN), interact with the physical world through Field Communication Interfaces (FCI), connecting to input/output (I/O) devices. The FCI supplies process values to the DCN, which then executes control actions based on these inputs and sends outputs back to the FCI. For a backup DCN to seamlessly assume the primary role, it must detect the primary's failure and resume the primary role with the former primary's last known state. The primary cyclically replicates its latest state to the backup and sends a heartbeat, i.e., a message with predetermined intervals for failure detection. Heartbeat absence signifies a possible primary failure. Controller redundancy communication is conventionally carried out over a dedicated, point-to-point connection [27, 18, 20], as illustrated in Figure 1. Failure of the redundancy link can partition the DCN pair, disrupting synchronization and causing their internal states to diverge. This divergence might result in inconsistent outputs to the FCI.

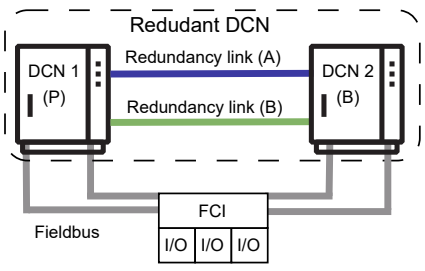


Figure 1: A redundant DCN (controller) pair synchronized with dedicated, redundant redundancy link.

Two strategies are common when managing failures in redundancy communication links: (i) disabling redundancy following

F. Lang, M. Volk (Eds):
Models for Formal Analysis of Real Systems (MARS 2024)
EPTCS 399, 2024, pp. 169–191, doi:10.4204/EPTCS.399.8

the failure of one of the links or (ii) continuing in redundant mode. These strategies reflect the alternatives a distributed system has in case of partitioning: remain consistent and sacrifice availability or vice versa—consequence of the Consistency, Availability, and Partitioning tolerance (CAP) theorem [6].

Disabling redundancy after a redundancy link failure compromises availability, as the backup won't activate if the primary controller fails before the link is repaired. While this method prioritizes consistency, a concurrent loss of both redundancy links can still lead to a dual primary situation [20].

The alternative, operating redundantly with only one functioning redundancy link, risks causing a dual primary situation if the remaining redundancy link fails. This is because the backup can not distinguish missing heartbeats due to a failure of the link from a failure of the primary. Some vendors call a dual primary scenario non-synchronized active units, signifying the consistency compromise following from CAP [18]. Controllers unable to communicate can not synchronize, leading to an inconsistent state in the redundant pair.

The advent of Industry 4.0 is steering industrial controllers towards a network-centric design [2, 4, 16]. As defined by the Open Process Automation Forum (OPAF), the DCNs and FCI are integrated into a cohesive communication network. Additionally, this network backbone can support redundancy communication and replace the redundancy link shown in Figure 1 with a network, see Figure 2.

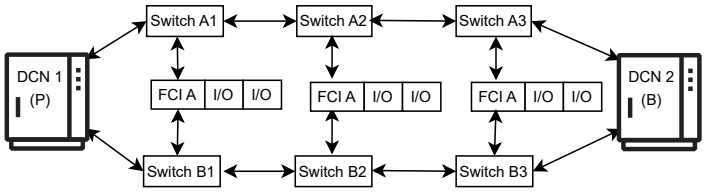
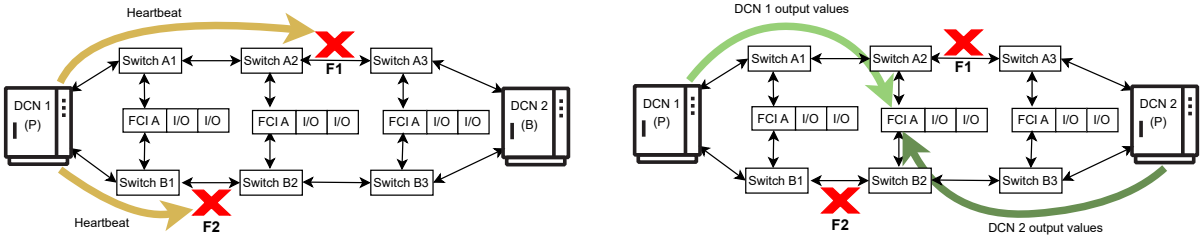


Figure 2: Redundant controllers connected over a redundant, disjoint network backbone.

When communication between a redundant DCN pair fails, as shown in Figure 3a, traditional approaches either disable redundancy at the first failure (F1) or allow the system to operate in a non-synchronized dual-primary mode, as shown in Figure 3b. Johansson et al. [10] introduce the Network Reference Point Failure Detection (NRP FD) for such redundant DCN systems. NRP FD prioritizes consistency while reducing the impact on availability. It uses an external Network Reference Point (NRP) as a tiebreaker for primary role determination, aiding the backup DCN in differentiating between primary and network failures. For a DCN to attain and retain the primary role, it must maintain communication



(a) (b)

Figure 3: (a) F1 and F2 exemplify network failures partitioning the redundant controller pair, preventing the heartbeat (and other communication) between DCN 1 and DCN 2. (b) Due to F1 and F2 caused partitioning, both DCN 1 and DCN 2 become primary and drive potentially inconsistent outputs.

with the NRP. The importance of addressing dual primary risks is emphasized in manuals recommending spatially separated redundancy links in current systems to avoid simultaneous damage and undefined system states [20].

To design an algorithm that guarantees the uniqueness of the primary the following questions need to be answered:

- How should the backup know about a failure?
- When should the backup become a primary?

As described by Johansson et al. in [10], the NRP FD uses heartbeats for primary failure detection (heartbeat) and a separate message for NRP reachability testing and detecting network failure. This introduces a potential vulnerability: the absence of a heartbeat is a sign of the primary failure, while NRP reachability is verified separately. Consequently, temporary disturbances could lead to inconsistencies, underscoring the importance of testing with temporal disturbances. Hence, one other question also have to be answered:

- How should we take care of the transient errors in switches or DCNs?

Since nondeterministic behavior is generally undesirable in control systems, particularly in high-integrity systems crucial for safety-critical solutions, like the ABB AC 800M High Integrity system [1], we need assurance of the correctness of the algorithm. Therefore, this paper describes in detail the modeling and formal verification of the NRP FD algorithm, considering the main safety property of "NoDualPrimary". We use Timed Rebeca which is an actor-based modeling language for reactive and distributed systems and its model checker tool Afra to model and verify NRF FD. We model different failures including transient errors and illustrate the results. We also propose an enhanced lease-based version of NRP FD that ensures a singular primary in the case of transient errors.

2 Network Reference Point Failure Detection (NRP FD) Algorithm

NRP FD targets failure detection in redundant controller pairs. In a standard system, two controllers, DCN 1 and DCN 2, function as primary and backup, respectively, as illustrated in Figure 4. The primary is unique in the system and interacts with I/O devices, while the backup, in standby mode, activates only upon primary failure. This concept is known as standby redundancy [21]. These controllers, DCN 1 and DCN 2, require communication, typically through a network facilitated by switches [4, 16]. Redundant controllers are often paired with dual independent networks for enhanced reliability, as depicted in Figure 4.

NRP FD is a heartbeat-based failure detection algorithm where the primary controller sends regular heartbeat messages to the backup via the networks connecting the redundant DCN pair [10]. These heartbeats, a push-based failure detection method, involve the primary sending messages to the backup at a known interval [19]. NRP FD differs from traditional heartbeat-based failure detection due to its NRP usage. An NRP must meet two requirements: (i) it should not share common cause failures with the redundant DCN pair, and (ii) be accessible from only one DCN in case of network partitioning. Each controller typically has one NRP candidate per independent network, as illustrated in Figure 4, where network switches serve as potential NRPs. The uper network in Figure 4 includes three switches $Switch\ A1$, $Switch\ A2$, and $Switch\ A3$, and the lower network includes $Switch\ B1$, $Switch\ B2$, and $Switch\ B3$. The NRP candidate set for the primary is $\{Switch\ A1, Switch\ B1\}$ and for the backup is $\{Switch\ A3, Switch\ B3\}$, and $Switch\ A1$ is the NRP.

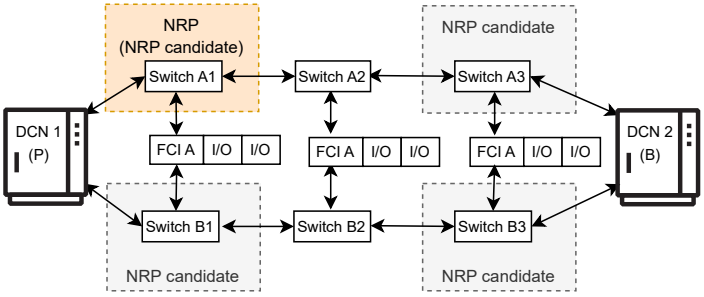


Figure 4: The redundant network backbone with the NRP and NRP candidates highlighted.

The operational procedure of NRP FD is as follows: before enabling redundancy, the primary DCN selects an NRP from the available NRP candidates. The heartbeat message communicates the NRP selection to the backup. The primary continuously monitors the NRP, ensuring its accessibility and proposing a change to the backup if the NRP is unreachable. If the backup doesn't acknowledge this change within a set time, the primary leaves the primary role. Concurrently, the backup continuously monitors heartbeats from the primary. If these are missing for a predetermined duration, the backup assesses its NRP connection. Should this connection be active, the backup takes the primary role. The following section will provide more details of the algorithm and its Timed Rebecca model.

3 Modeling and Verification of NRP FD using Timed Rebeca

We use Timed Rebeca language and its integrated model checker tool, Afra, to model and verify NRP FD. For modeling NRP FD, we have used the description of the protocol and the diagrams provided in [10] as well as several meetings with the industrial partners to clarify the details and choose the appropriate level of abstraction, which we will discuss in the remainder of this section.

3.1 The actor-based language, Timed Rebeca

Rebeca (Reactive Object Language) [26, 22] is an actor-based language designed for modeling and formal verification of reactive concurrent and distributed systems. Actors [8, 3] are units of concurrency. 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 a 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, sharing the same name as the class, 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) [24, 12] 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 at the time of 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 [25]. 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 [14]. If more than one actor or event are enabled at the same time, then the model checker builds all the possible execution traces, using priorities you can cut some of the branches.

3.2 Modeling NRP-FD in Timed Rebeca

We model Figure 4 using Timed Rebeca. The model is extensible meaning that the number of switches and nodes can be increased. In the Timed Rebeca model each node and each switch is modeled as an actor, their communication is modeled as message passing, and reactions to each message, signal, and timed event are modeled using message servers. A Rebeca model includes reactive class definitions, defining the behavior of the rebecs (actors) within the model. L1[1] illustrates some parts of the Timed Rebeca model for NRP FD.

In the NRP-FD model, we have two different element types, Node and Switch. Each element type is defined as a reactive class, *Node* (L1, line 10) and *Switch* (L1, line 40). Each reactive class has a constructor. A constructor is a unique method which is called when the actor is instantiated. Initialization of the variables is done in the constructor. We instantiate two nodes with ids 100 and 101 and six switches (A1-A3 and B1-B3) in the main section (L1, lines 59-68). A node can be a primary or a backup, and a switch can be a non-terminal switch (not connected to a DCN), an NRP candidate, or an NRP. Each node has an NRP candidate (switch) for each network, i.e., switches A1 and B1 with ids 1 and 4, respectively for DCN1 and switches A3 and B3 for DCN2 with ids 3 and 6, respectively (L1, lines 66-67). The parameters in the instantiation statements are used to set different types and also pass other necessary information to the constructor.

We select DCN1 with id 100 as the primary at the beginning of the algorithm (second parameter in lines 66-67 of L1). There are two known rebecs in the reactive class *Node*, meaning it can send messages to these rebecs. We have a method call in the constructor of the *Node*, i.e., *runMe* (L1, line 22). In *runMe* (L1, line 28) the DCN checks its state using the state variable *mode* and then serves the corresponding behavior (L1, lines 30-34). Note that, the last line of *runMe* (L1, line 35) is a self-call followed by an after with *heartbeat_period* as its parameter, modeling a periodic event, i.e., "$runMe()after(heartbeat\_period)$;". It means that in every *heartbeat_period* (determined in the code L1, line 1), *runMe* is executed. The *heartbeat_period* should be significantly larger than other timing parameters. This is because all events must be handled during a heartbeat interval. Regarding timing

[1] We use L1, L2 and L3 to refer to Listing 1, Listing 2 and Listing 3, respectively.

```
env int heartbeat_period = 1000;
env int max_missed_heartbeats = 2;
env int ping_timeout =500;
env int nrp_timeout = 500;
env byte NumberOfNetworks = 2;
env int switchA1failtime = 2500;
...
env int networkDelay = 1;
env int networkDelayForNRPPing = 1;
reactiveclass Node (4){
    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 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 endSwitch , Switch sw1, Switch sw2, int myFailTime) {
       mynetworkId = networkId;
       id = myid;
       terminal=endSwitch;
       amINRP = false;
       failed = false;
       switchTarget1 = sw1;
       switchTarget2 = sw2;
       ...
    }
    msgsrv switchFail(){ failed = true; amINRP=false;}
    msgsrv pingNRP_response(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(DCN1):(1, 0, true , switchA2 , switchA2 , switchA1failtime);
    @Priority(1) Switch switchA2(DCN1):(2, 0, false , switchA1 , switchA3 , switchA1failtime);
    @Priority(1) Switch switchA3(DCN2):(3, 0, true , switchA2 , switchA2 , switchA3failtime);
    @Priority(1) Switch switchB1(DCN1):(4, 1, true , switchB2 , switchB2 , switchB1failtime);
    @Priority(1) Switch switchB2(DCN1):(5, 1, false , switchB1 , switchB3 , switchB1failtime);
    @Priority(1) Switch switchB3(DCN2):(6, 1, true , switchB2 , switchB2 , switchB3failtime);
    @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 (Full version in Appendix C).

parameters in modeling, we carefully consider values so that the model matches the reality. We will discuss more on timing in the following.

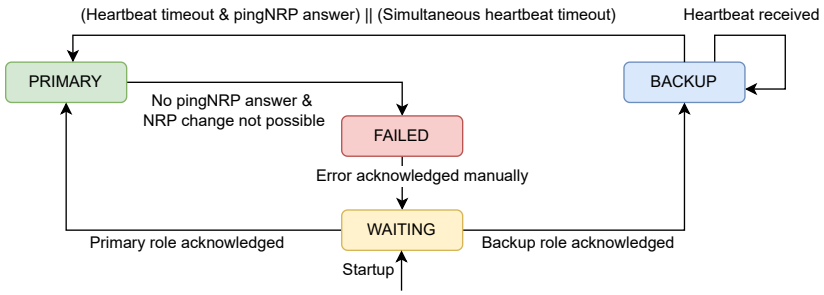


Figure 5: Different modes of a DCN in NRP FD in the Rebeca model. *WAITING* is the initial mode. The node transitions from *WAITING* to *PRIMARY* or *BACKUP* based on the value passed to its constructor. From *PRIMARY*, it moves to *FAILED* if after sending a *pingNRP* it receives no response from NRP within the deadline, and it cannot change the NRP either. In the *BACKUP* state, the node transitions to *PRIMARY* if the heartbeat timeouts and *pingNRP* detects a responsive NRP, or when the heartbeat timeout occurs simultaneously for both networks. In the latter case the backup node assumes that the primary node failed, because it is unlikely that there is a failure in both networks. The node stays in *BACKUP* mode as long as it is receiving heartbeats. It remains in *FAILED* until the situation is resolved manually.

In NRP FD, DCNs have four modes, *WAITING*, *BACKUP*, *PRIMARY*, and *FAILED*, as detailed in the diagram in Figure 5. In the Rebeca model, we set the initial mode of DCN to *WAITING* in the constructor of *Node*, L1, line 19. We pass the primary id to both nodes and in the *WAITING* mode the variable denoting the role is set accordingly, and an NRP is announced.

In the *PRIMARY* mode, the primary DCN tests the NRP reachability with *pingNRP*, i.e., sends message *pingNRP* to the NRP which then is served using the message server *pingNRP* (L1, line 55). In a real system, the *pingNRP* could be realized with an Internet Control Message Protocol (ICMP) echo (commonly known as ping) or another suitable protocol depending on the NRP's capabilities. If the NRP fails to respond, the primary announce a new NRP, assuming alternatives are available (using *new_NRP* message server, L1, line 56). After assuring that an NRP exists, the primary DCN sends heartbeats. If there is no available NRP, the primary transition to the *FAILED* mode (*ping_timed_out* in L1, line 25).

In the *BACKUP* mode, the DCN expects heartbeats from the primary. The heartbeat period and tolerance limits (i.e., the number of missed heartbeats before a timeout is declared) must be carefully set to minimize false positives due to transient disturbances. Given that typical DCN redundancy involves two disjoint network paths, a heartbeat is expected on each network path per period. Simultaneous timeouts on all paths likely indicate a primary failure rather than failure of both networks. Thus, NRP FD offers an optimization: transitioning directly to the *PRIMARY* mode upon simultaneous heartbeat timeouts, bypassing the *pingNRP* exchange. However, this optimization slightly increases the risk of dual primaries. This is a bug that model checking catches. The number of maximum missed heartbeats is set to 2 (*max_missed_heartbeat* in L1, line 2). L2, shows the *BACKUP* part of the message server *runMe*. The variables *heartbeats_missed*_1 and *heartbeats_missed*_2 are counters for heartbeats on the two networks which will increase at each period, and is reset to zero when a heartbeat is received.

The backup DCN counts consecutive *heartbeats_missed* for each network. If both counters exceed the defined limit of *max_missed_heartbeat* (L2, line 4), the backup detects a failure and sends a *pingNRP* to the NRP to verify its reachability. If the NRP is reachable, the DCN transitions from *BACKUP* to the *PRIMARY* state (in *ping_timed_out*, L1, line 25).

In the *FAILED* mode, NRP FD awaits the acknowledgment that manually confirms the resolution of the issues that triggered the transition to *FAILED*.

```
1  case 2: //BACKUP :
2   heartbeats_missed_1++;
3   heartbeats_missed_2++;
4   if (heartbeats_missed_1 > max_missed_heartbeats && heartbeats_missed_2 > max_missed_heartbeats){
5    if(heartbeats_missed_1==heartbeats_missed_2 && heartbeats_missed_2==max_missed_heartbeats+1){
6     mode = PRIMARY;
7     primary=id;
8     ...
9    }else{
10    heartbeats_missed_1 =
       ↪  (heartbeats_missed_1>max_missed_heartbeats+2)?max_missed_heartbeats+2:heartbeats_missed_1;
11    heartbeats_missed_2 =
       ↪  (heartbeats_missed_2>max_missed_heartbeats+2)?max_missed_heartbeats+2:heartbeats_missed_2;
12    if(NRP_network==0){
13     ping_pending = true;
14     NRP_network=-1;
15     out1.pingNRP(id, NRP_switch_id) after(5);
16     ping_timed_out() after(ping_timeout);
17    }else{ ...  // the other network }
18    NRP_pending = true;
19   }
20  }
21  else if(heartbeats_missed_1 > max_missed_heartbeats || heartbeats_missed_2 >
     ↪  max_missed_heartbeats){...}
```

Listing 2: (L2) The behavior of a DCN in *BACKUP* mode, in the message server *runMe* (full version is provided in Appendix C).

Accuracy of the model. Based on the real situation, we consider the topology and the way the DCNs interact with each other. The rationale for tolerating up to two lost heartbeats ($max\_missed\_heartbeats = 2$) is based on the low bit error rate of gigabit Ethernet and the ability of a heartbeat message to fit within a standard 1500-byte Ethernet frame. This suggests a low likelihood of losing heartbeat messages, especially across two disjoint networks, thus minimizing the risk of false positives due to regular disturbances. The *heartbeat_period*, combined with *max_missed_heartbeats*, determines the reaction time - the duration from the occurrence of a primary failure to the point at which the backup takes over the primary role. The takeover time requirement varies by domain; for process control, a maximum of 500 milliseconds is tolerable, as suggested by Hegazy et al. [7]. System manuals indicate feasible heartbeat periods are in the tens of milliseconds range[20, 18]. Regarding propagation and *pingNRP* response times, the propagation of a full-sized Ethernet frame on Gigabit Ethernet is about 12 microseconds, negligible compared to the heartbeat period. The NRP's response time is implementation-dependent, potentially under a millisecond. If ICMP ping is employed, a few milliseconds response times are achievable [10]. We've defined the *heartbeat_period* as 1000 time units and set the *ping_timeout* and *nrp_timeout* to 500 time units. We also consider *networkDelay* and *networkDelayForNRPPing* as 1 unit of time. We use the keyword $after$ when DCNs ping the NRP node and set it to 5 units of time. These values are chosen to be approximately close to the actual values and preserve the sequence of the messages. Therefore, they may vary, for instance, to a greater or lesser extent. But all timing events should be handled within one

period, 1000 time units in our model. We used the $after$ construct where we needed to respect the order of execution.

4 Model checking of NRP FD using Afra

We can define our desired properties using assertions in a separate file in Afra and perform model checking. A snapshot of Afra is provided in Appendix B. The main safety property, "*NoDualPrimary*," is shown in L3. This property is set to recognize the dual primary state, i.e., in no state the modes of the two DCNs are both primary. We first define a set of atomic propositions, and then the assertions based on these propositions. Timed Rebeca has a TCTL model checking but it is not integrated in Afra. In many cases, looking at the visualization of the state space helps us see the problems with the algorithm.

```
1  property {
2          define {
3                  DCN1Primary = (DCN1.mode ==1);
4                  DCN2Primary = (DCN2.mode ==1);
5          }
6  Assertion{ NoDualPrimary:!(DCN1Primary && DCN2Primary); }}
```

Listing 3: The safety property "*NoDualPrimary*" for NRP FD.

For model checking, we consider the regular system behaviour, and scenarios where we have failures of DCNs and switches. We examine all the possible failure combinations of DCNs and switches at the start of handling an event, and perform model checking to provide a comprehensive analysis. We have modeled failures in three scenarios each of which can have different cases:

1. Failures on each event. In this scenario, we add the following commands at the beginning of each message server for DCNs and switches, simulating the possibility of their failure. Since this scenario models the failure where an event should be handled, we refer to it as event-based. The expression "$?(true, false)$" represents a nondeterministic choice between true and false. When the value true is chosen then a variable is set, this variable is checked in the beginning of the messages servers and if it is set the message server is not executed.

```
//Possible failure for a DCN:
if(?(true,false)) nodeFail();
//Possible failure for a Switch:
if(?(true,false)) switchFail();
```

2. Failures that occur at specific times. We define a set of variables to model the failure of different DCNs and switches at specific times. By manipulating these variables, we can model various combinations of DCN and switch failures at different times across multiple model checking runs.

```
env int switch1failtime = 0; env int switch2failtime = 2500; env int switch3failtime = 0;
env int switch4failtime = 2500;
env int node1failtime = 0; env int node2failtime = 0;
..
//Failure of a DCN at a specific point of time. Value zero means no failure.
if(myFailTime!=0) nodeFail() after(myFailTime);
...
//Failure of a Switch at a specific point of time. Value zero means no failure.
if (myFailTime!=0) switchFail() after(myFailTime);
```

3. Transient failures. These failures could occur, for example, if an attacker deliberately drops the heartbeats for more than the maximum allowed misses (*max_missed_heartbeats*) on both networks.

Subsequently, the backup DCN, upon detecting missed heartbeats, checks the NRP. If the NRP is reachable, it becomes the primary, assuming that the primary has failed, resulting in a dual-primary situation. So we model a transient failure where both heartbeats are missed. Part (not all) of the code for this scenario is the following, which states that only if we do not have an attacker, then the heartbeats will be sent.

```
if(attacker<1){
    out1.heartBeat(0, id) after(networkDelay);
    out2.heartBeat(1, id) after(networkDelay);
}
```

Table 1 illustrates the scenarios we have considered and checked. Number of states and transitions are also reported. Note that in the cases where the the assertion is violated, model checking is stopped after reaching a counter example. Case 1 is the case with no failure. Case 2 is the event-based failure scenario where we investigate all combinations of failures for any DCN or switch, where they stop reacting to the events. Cases 3 to 5 consider failures at time 2500 for DCN1, switchA1 and switchA3, respectively. This number is intended to go through a full round of algorithm execution, with two heartbeats. We have considered case 3 for the *PRIMARY* failure as DCN1 is initially set as the primary DCN. We also consider cases 4, 6, and 7 as failures of switches A1 and B1 can cause the primary DCN to be disconnected from the networks. Case 5 is also considered to model a situation where the backup cannot ping the NRP. Case 8 is modeling the transient error. There are three cases where the model violates the property.

Table 1: Different test scenarios, without any failures, and with different types of failures

Case	Configuration for failures	Result	no. of states and transitions
1	Without failure	✓	38, 49
2	Failures on each event	✗	3539, 4677
3	DCN1 fails at time 2500	✓	113, 138
4	switchA1 fails at time 2500	✓	114, 134
5	switchA3 fails at time 2500	✓	146, 179
6	switchA1 fails at time 2500 and switchB1 at time 3500	✓	187, 223
7	switchA1 and switchB1 fails simultaneously at time 2500	✗	70, 88
8	Heartbeats are missing because of transient errors	✗	35, 42

Afra generates a counter-example in cases of any violation (here for cases 2, 7 and 8 of Table 1). We can explore the states in the counter-example and see the value of the state variables in each of them. A snapshot of the state space showing the dual primary situation for the case 7 is depicted in Figure 8 of Appendix D. These cases may be rare situations in reality, but in formal verification we detect and eliminate the corner cases. To overcome these issues, we provide an extension for NRP FD, which will be described next.

4.1 Leasing NRP FD

To address failure issues, we provide an enhanced NRP FD version called Leasing NRP FD. First, we remove the optimization, i.e., transitioning directly from *BACKUP* to the *PRIMARY* mode upon simultaneous heartbeat timeouts, bypassing the *NRPPing* exchange (L2, lines 5-9).

While NRP FD prioritizes consistency, even without optimization, there remains a non-zero probability of failure. The heartbeat and *pingNRP* messages are separate: the heartbeat indicates whether the primary is alive, and the *pingNRP* informs the backup about its separation from the NRP or the NRP's failure. Since these messages are distinct and can be independently disrupted, it's theoretically possible, as indicated by verification, that a temporary disturbance might disrupt the heartbeats. This

disruption could lead the backup to believe the primary has failed, and upon a successful *pingNRP* following the transient disturbance, it might erroneously become the *PRIMARY*, even while the other DCN remains primary. To address this vulnerability, we introduce the Leasing NRP FD, where the primary role is 'leased' from the NRP. This leasing can be implemented in various ways. In our model, the NRP timestamps the latest *pingNRP* from the primary, and then the backup checks this timestamp. Full version of Leasing NRP FD is provided in Appendix C and also on the Rebeca GitHub page[2]. Even with a low probability of dual primary occurrences in the original NRP FD, this inherent algorithmic trait could lead to nondeterministic behavior, which is unacceptable in safety-critical solutions. Thus, there's a need for algorithms like Leasing NRP FD, which eliminate such violations and are more suitable for safety-critical systems. For this new algorithm, Afra created 15891 states, and 34053 transitions, and the assertion is satisfied.

5 Why Timed Rebeca?

In [23], Sirjani argues that when selecting a modeling language, expressiveness is a key factor, but faithfulness to the system being modeled and usability for the modeler are equally crucial. Faithfulness is about how similar the model and the system are. It determines if and how the structures and features supported by the modeling language match with the requirements of the system's domain. Faithfulness makes reusability possible, also in cases gives us better analyzability and traceability. Usability concerns the modeler, and how swiftly the modeler can use the language. These two aspects together are called as friendliness in [23].

Timed Rebeca is a language for modeling asynchronous communication in distributed systems, incorporating a focus on time-related aspects. Regarding faithfulness, actors are units of concurrency like the controllers and switches in our case study. Timed Rebeca is event-driven, taking messages/events from the message/event bags and executing their corresponding message servers. Timed Rebeca is used for modeling and verification in many domains including different network protocols, schedulability in sensor networks and Network on Chip (NoC) [23]. Considering our problem in the domain of distributed control systems, Timed Rebeca provides a natural mapping of structures, features, and flow of control for our purpose such as modeling the topology of the network, behavior of the DCNs and switches based on their roles, the way they communicate using message passing, progress of time required for handing a message, network delay, and periodic events using primitive timing keywords. Message queues/buffer are not explicit and the modeler does not need to manage them. Timing concept is intuitive, and you model the behavior from the perspective of each actor.

Regarding usability, it has a structure like a programming language, hence, it is easy for programmers to use. Debugging can be done based on the counterexamples and going through the model checking process iteratively. Timed Rebeca is supported by an Eclipse IDE called Afra [11]. Afra provides a model checker tool for the family of Rebeca languages. The modeler enters the model and the properties in separate files, then model check and debug the model in Afra. Timed models result in an infinite number of states in the state space due to the progress of time, leading to unbounded transition systems. A shift-equivalence relation is introduced for Timed Rebeca in [12, 13] to ensure a bounded state space. Afra utilizes this relation to generate the state space including local actor states and logical time. Desired properties can be written as assertions in a separate file in Afra. In case of violation, a counter-example is shown visually alongside the model which gives us the ability to traverse and check the values of the actors' variables. As the state space is provided in an XML file, it is also possible to have a visual

[2]https://github.com/rebeca-lang

representation of the entire state space (see an example in Figure 8, App. D). All the above gives us a natural and easy way to model our system, and also provide us analzability and traceability.

6 Related work

Control systems evolve from hierarchical, controller-centric structures toward a flatter, network-centric architecture, enhancing interconnectivity and facilitating communication with cloud services and edge devices [4, 16]. These advancements have been leveraged for fault tolerance—employing backup DCNs in the cloud or orchestrators to recover from DCN failures [7, 9]. To our knowledge, the NRP FD algorithm is the first effort to reduce the CAP theorem's [6] availability tradeoff while preserving consistency in DCN redundancy scenarios [10]. The tradeoff mandated by the CAP theorem is evident in today's redundant DCN systems. Control system user manuals concretize the tradeoff with the different approaches described, which either strive to maintain consistency or prioritize availability upon redundancy link failure [20, 18]. Fault tolerance is ensured using duplicate links, as depicted in Figure 1. With duplicated links, consistency can be prioritized by disabling DCN redundancy if one link fails [20]. However, a dual primary situation arises if both links fail simultaneously. Vice versa, availability is prioritized by not disabling redundancy upon one link failure [18]. The Leasing NRP FD version assures consistency by maintaining a single primary in all failure scenarios.

Appointing a primary is a leader election problem, and various leader election algorithms exist, such as the well-known Bully algorithm [5]. However, the Bully algorithm, and variants thereof, elects multiple leaders in networking partitioning situations, one leader per partition. Alternatively, consensus protocols like Raft and Paxos require a majority [17, 15], ensuring consistency even when partitions occur, as only the majority-containing partition progresses. However, the most common DCN redundancy configurations, typically comprising a primary and a backup, do not allow a majority to form in the event of a partition separating the DCNs [21]. The NRP FD method introduces the NRP that, in combination with a DCN, establishes a majority [10]. The NRP could be as simple as a layer two network switch responding to an ICMP Ping, providing a means to favor consistency over availability. This paper describes the modeling and verification of the NRP FD strategy, along with a novel, lightweight enhancement ensuring a single primary, i.e., guaranteeing that consistency is preserved due to more than one DCN taking the primary role. The algorithm is being extended in different directions, considering different configurations and features. Our aim is to enrich our model align with the extensions of NRP FD, when the extensions are available.

7 Conclusion and Future Work

In this paper we describe the process of modeling and formal verification of NRP FD protocol which is used for preserving consistency in DCN redundancy scenarios using Timed Rebeca and Afra. We investigate different failure scenarios and identify situations where network partitioning can lead to a dual primary. We propose an extension, Leasing NRP FD, which preserves consistency and ensures robustness against different failures. For future research, we focus on the extensibility and flexibility of the proposed protocol including the exploration of a dynamic network topology, multiple backups and multiple primaries. The latter could be a redundancy plan with a single backup for multiple primaries, each with different and unique characteristics such as specific heartbeat time and network delay. Additionally, we aim to incorporate probability considerations rather than just focusing on the possibility

(of failures). As another future direction, we plan to investigate the availability trade-off. While NRP-FD prioritizes consistency, this may result in compromising availability. Quantifying this trade-off is a potential direction for further research.

Acknowledgment

We acknowledge the support of the Swedish Knowledge Foundation via the synergy project SACSys (Safe and Secure Adaptive Collaborative Systems) and the Profile DPAC (Dependable Platforms for Autonomous Systems and Control). We also acknowledge the support of the Swedish Foundation for Strategic Research (SSF) via the Serendipity project.

References

[1] *AC 800M High Integrity*. https://new.abb.com/control-systems/safety-systems/system-800xa-high-integrity/ac-800m-hi-controller. Accessed: 2024-03-07.

[2] *The DCS of Tomorrow - ABB's Process Automation System Vision Whitepaper*. https://new.abb.com/control-systems/control-systems/envisioning-the-future-of-process-automation-systems/automation-system-whitepaper. Accessed: 2024-03-07.

[3] Gul Agha (1986): *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, doi:10.7551/mitpress/1086.001.0001.

[4] Johan Åkerberg, Johan Furunäs Åkesson, Jorgen Gade, Maryam Vahabi, Mats Björkman, Mehrzad Lavassani, Rahul Nandkumar Gore, Thomas Lindh & Xiaolin Jiang (2021): *Future industrial networks in process automation: Goals, challenges, and future directions*. Applied Sciences 11(8), p. 3345, doi:10.3390/app11083345.

[5] H. Garcia-Molina (1982): *Elections in a Distributed Computing System*. IEEE Trans. Comput. 31(1), pp. 48–59, doi:10.1109/TC.1982.1675885.

[6] Seth Gilbert & Nancy Lynch (2002): *Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services*. Acm Sigact News 33(2), pp. 51–59, doi:10.1145/564585.564601.

[7] T. Hegazy & M. Hefeeda (2015): *Industrial Automation as a Cloud Service*. IEEE Trans. Par. and Distr. Syst. 26(10), pp. 2750–2763, doi:10.1109/TPDS.2014.2359894.

[8] Carl Hewitt, Peter Bishop & Richard Steiger (1973): *A universal modular actor formalism for artificial intelligence*. In: Proceedings of the 3rd international joint conference on Artificial intelligence, Morgan Kaufmann Publishers Inc., pp. 235–245. Available at http://ijcai.org/Proceedings/73/Papers/027B.pdf.

[9] Bjarne Johansson, Mats Rågberger, Thomas Nolte & Alessandro V Papadopoulos (2022): *Kubernetes orchestration of high availability distributed control systems*. In: IEEE Int. Conf. on Ind. Tech. (ICIT), doi:10.1109/ICIT48603.2022.10002757.

[10] Bjarne Johansson, Mats Rågberger, Alessandro Papadopoulos & Thomas Nolte (2023): *Consistency Before Availability: Network Reference Point based Failure Detection for Controller Redundancy*. In: 28th International Conference on Emerging Technologies and Factory Automation, pp. 1–8, doi:10.1109/ETFA54631.2023.10275664.

[11] Ehsan Khamespanah, Marjan Sirjani & Ramtin Khosravi (2023): *Afra: An Eclipse-Based Tool with Extensible Architecture for Modeling and Model Checking of Rebeca Family Models*. In Hossein Hojjat & Erika Ábrahám, editors: Fundamentals of Software Engineering, Springer Nature Switzerland, Cham, pp. 72–87, doi:10.1007/978-3-031-42441-0_6.

[12] Ehsan Khamespanah, Marjan Sirjani, Zeynab Sabahi-Kaviani, Ramtin Khosravi & Mohammad-Javad Izadi (2015): *Timed Rebeca schedulability and deadlock freedom analysis using bounded floating time transition system*. *Science of Computer Programming* 98, pp. 184–204, doi:10.1016/j.scico.2014.07.005.

[13] Ehsan Khamespanah, Marjan Sirjani, Mahesh Viswanathan & Ramtin Khosravi (2015): *Floating time transition system: more efficient analysis of timed actors*. In: *Formal Aspects of Component Software*, Springer, pp. 237–255, doi:10.1007/978-3-319-28934-2_13.

[14] Ramtin Khosravi, Ehsan Khamespanah, Fatemeh Ghassemi & Marjan Sirjani (2024): *Actors Upgraded for Variability, Adaptability, and Determinism*. In: *Workshop on State-of-the-Art of Active Objects*, pp. 226–260, doi:10.1007/978-3-031-51060-1_9.

[15] Leslie Lamport (2001): *Paxos Made Simple*. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58. Available at `https://www.microsoft.com/en-us/research/publication/paxos-made-simple/`.

[16] Björn Leander, Bjarne Johansson, Tomas Lindström, Olof Holmgren, Thomas Nolte & Alessandro V Papadopoulos (2023): *Dependability and Security Aspects of Network-Centric Control*. In: *2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*, IEEE, pp. 1–8, doi:10.1109/ETFA54631.2023.10275344.

[17] Diego Ongaro & John Ousterhout (2014): *In search of an understandable consensus algorithm*. In: *2014 USENIX annual technical conference (USENIX ATC 14)*, pp. 305–319. Available at `https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro`.

[18] PACSys (2023): *PACSystems™ RX3i Hot Standby CPU Redundancy*. `https://emerson-mas.my.site.com/communities/en_US/Documentation/PACSystems-Hot-Standby-CPU-Redundancy-Users-Manual`. Accessed: 2024-03-07.

[19] Benjamin Satzger, Andreas Pietzowski, Wolfgang Trumler & Theo Ungerer (2007): *A new Adaptive Accrual Failure Detector for Dependable Distributed Systems*. In: *In ACM Symposium on Applied Computing (SAC 2007*, pp. 551–555, doi:10.1145/1244002.1244129.

[20] Siemens (2024): *Siemens System Manual S7-1500R/H redundant system*. `https://cache.industry.siemens.com/dl/files/833/109754833/att_965668/v3/s71500rh_manual_en-US_en-US.pdf`. Accessed: 2024-03-07.

[21] Andrei Simion & Calin Bira (2023): *A review of redundancy in PLC-based systems*. *Advanced Topics in Optoelectronics, Microelectronics, and Nanotechnologies XI* 12493, pp. 269–276, doi:10.1117/12.2644462.

[22] Marjan Sirjani (2006): *Rebeca: Theory, Applications, and Tools*. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf & Willem P. de Roever, editors: *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* 4709, Springer, pp. 102–126, doi:10.1007/978-3-540-74792-5_5.

[23] Marjan Sirjani (2018): *Power is Overrated, Go for Friendliness! Expressiveness, Faithfulness, and Usability in Modeling: The Actor Experience*. In Marten Lohstroh, Patricia Derler & Marjan Sirjani, editors: *Principles of Modeling - Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*, *Lecture Notes in Computer Science* 10760, Springer, pp. 423–448, doi:10.1007/978-3-319-95246-8_25.

[24] Marjan Sirjani & Ehsan Khamespanah (2016): *On Time Actors.* In Erika Ábrahám, Marcello M. Bonsangue & Einar Broch Johnsen, editors: *Theory and Practice of Formal Methods*, *Lecture Notes in Computer Science* 9660, Springer, pp. 373–392, doi:10.1007/978-3-319-30734-3_25.

[25] Marjan Sirjani, Edward A. Lee & Ehsan Khamespanah (2020): *Verification of Cyberphysical Systems*. *Mathematics* 8(7), doi:10.3390/math8071068.

[26] Marjan Sirjani, Ali Movaghar & MohammadReza Mousavi (2001): *Compositional Verification of an Object-Based Model for Reactive Systems*. In: *AVoCS 2001*. Available at `https://rebeca-lang.org/assets/papers/2001/CompositionalVerificationOfAnObject-BasedModelForReactiveSystems.pdf`.

[27] Jacek Stój (2020): *Cost-effective hot-standby redundancy with synchronization using EtherCAT and real-time ethernet protocols*. IEEE Transactions on Automation Science and Engineering 18(4), pp. 2035–2047, doi:10.1109/TASE.2020.3031128.

A Rebeca Syntax

An abstract syntax of Timed Rebeca is provided in Figure 6.

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

Figure 6: 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, the use of $\langle ... \rangle$ with repetition indicates a comma-separated list. Square brackets [...] indicate that the enclosed text is optional [23].

B Afra

A snapshot of Afra is provided in Figure 7. The state space statistics are shown in the bottom middle. The generated counterexample is shown in the top right of the panel. At the bottom right, we can see the value of the state variables in each selected state from the counterexample.

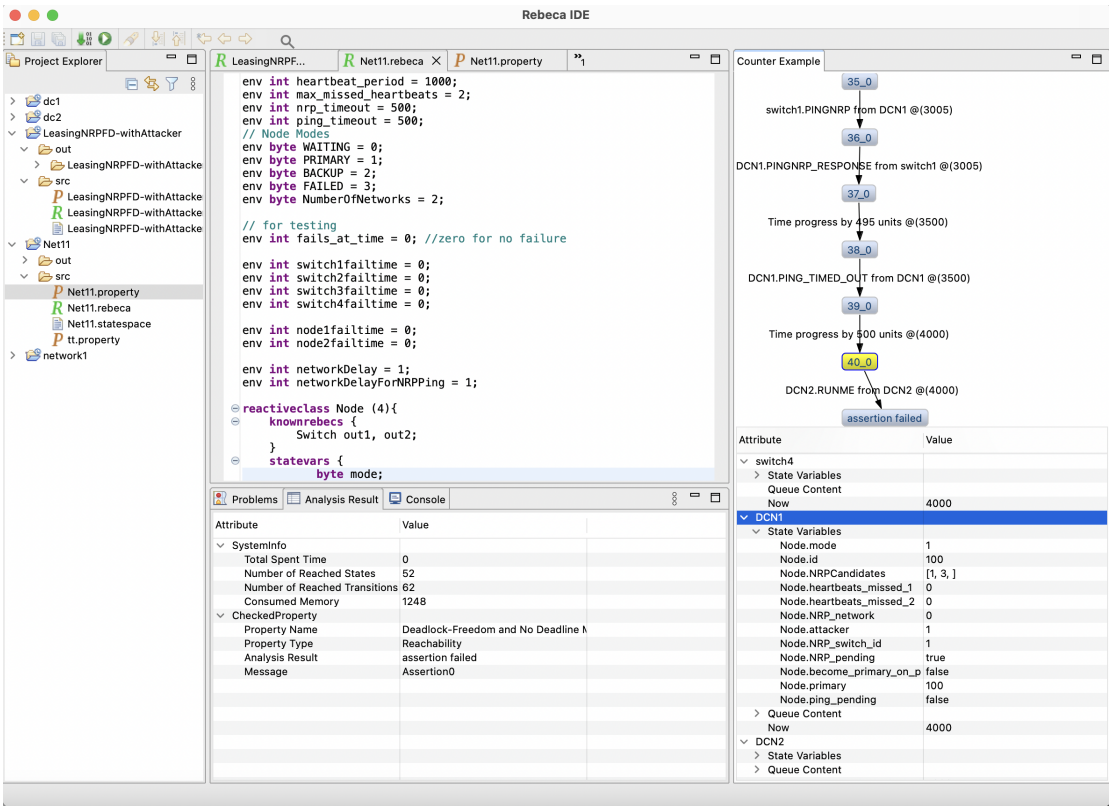


Figure 7: A snapshot of Afra.

C Timed Rebeca model of the Leasing NRP FD

In the following the Timed Rebeca model of the Leasing NRP FD is provided.

```
env int heartbeat_period = 1000;
env int max_missed_heartbeats = 2;
env int ping_timeout =100;
env int nrp_timeout = 100;
// Node Modes
env byte WAITING = 0;
env byte PRIMARY = 1;
env byte BACKUP = 2;
env byte FAILED = 3;
env byte NumberOfNetworks = 2;

env byte MAX_SWITCHES = 99;
// 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;
env int networkDelayForNRPPing = 1;

reactiveclass Node (4){
    knownrebecs {
        Switch out1, out2;
    }
    statevars {
        byte mode;
        int id;
        int [2] NRPCandidates;
        int heartbeats_missed_1;
        int heartbeats_missed_2;
        int NRP_network;
        int attacker;
        int which;
        boolean prevWhich;
        int NRP_switch_id;
        boolean NRP_pending;
        boolean become_primary_on_ping_response;
        int primary;
        boolean ping_pending;
        boolean init;
    }
    Node (int Myid, int Myprimary, int NRPCan1_id, int NRPCan2_id, int myFailTime) {
        id = Myid;
        attacker = 0;
        which=0;
        prevWhich=true;
        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;
        init=true;

        mode = WAITING;
        if(myFailTime!=0) nodeFail() after(myFailTime);
        runMe();
    }
```

```
71      msgsrv new_NRP_request_timed_out() {
72          // if(?(true,false)) nodeFail();
73          if (mode == BACKUP) {
74              if (NRP_pending) {
75                  NRP_pending = false;
76                  if (become_primary_on_ping_response)
77                      become_primary_on_ping_response = false;
78              }
79          }
80      }
81      // logical action ping_timed_out(ping_timeout)
82      msgsrv ping_timed_out() {
83          // if(?(true,false)) nodeFail();
84          if (mode == BACKUP) {
85              if (ping_pending) ping_pending = false;
86              else{
87                  if(which>1){
88                      mode = PRIMARY;
89                      heartbeats_missed_1 = 0;
90                      heartbeats_missed_2 = 0;
91                      primary=id;
92                      if(NRP_network==0) out1.new_NRPBack(id, id,NRP_network, NRP_switch_id);
93                      else out2.new_NRPBack(id,id, NRP_network, NRP_switch_id);
94                      mode = PRIMARY;
95                      heartbeats_missed_1 = 0;
96                      heartbeats_missed_2 = 0;
97                      primary=id;
98                      NRP_pending = true;
99                  }else NRP_pending = true;
100             }
101         }else if (mode == PRIMARY){
102             if (ping_pending){
103                 NRP_network++;
104                 if(NRP_network<NumberOfNetworks){
105                     NRP_switch_id = NRPCandidates[NRP_network];
106                     if(NRP_network==0) out1.new_NRP(id, id,NRP_network, NRP_switch_id);
107                     else out2.new_NRP(id,id, NRP_network, NRP_switch_id);
108                 } else {
109                     NRP_network=NumberOfNetworks;
110                     mode=  WAITING;
111                 }
112                 NRP_pending = true;
113             } else{
114                 if(attacker<1){
115                     out1.heartBeat(0, id) after(networkDelay);
116                     out2.heartBeat(1, id) after(networkDelay);
117                 }
118             }
119         }
120     }
121     msgsrv pingNRP_response(int mid, boolean w, boolean pw){
122         // if(?(true,false)) nodeFail();
123         if (mode==WAITING);
124         else if (mode == BACKUP){
125             if(!w && !pw) which++;
126             else which=0;
127             if(which>1)
128             ping_pending = false;
129         }
130         else if (mode == PRIMARY)
131             ping_pending = false;
132         else if (mode==FAILED);
133     }
134     msgsrv new_NRP(int mid,int prim, int mNRP_network, int mNRP_switch_id) {
135         // if(?(true,false)) nodeFail();
136         if(mode!= FAILED){
137             NRP_network = mNRP_network;
138             NRP_switch_id = mNRP_switch_id;
139         }
140     }
```

```
141     msgsrv new_NRPBack(int mid,int prim, int mNRP_network, int mNRP_switch_id) {
142         // if(?(true,false)) nodeFail();
143         if(mode!= FAILED){
144             NRP_network = mNRP_network;
145             NRP_switch_id = mNRP_switch_id;
146         }
147     }
148     msgsrv runMe(){
149       switch(mode){
150         case 0: //WAITING :
151             if(init){
152                 if (id == primary){
153                     mode = PRIMARY;
154                     NRP_network++;
155                     if(NRP_network<NumberOfNetworks){
156                         NRP_switch_id = NRPCandidates[NRP_network];
157                         if(NRP_network==0)out1.new_NRP(id,id, NRP_network, NRP_switch_id);
158                         else out2.new_NRP(id,id, NRP_network, NRP_switch_id);
159                     } else NRP_network=NumberOfNetworks;
160                 } else mode =BACKUP;
161                 init=false;
162             }
163             break;
164         case 1: //PRIMARY :
165             attacker++;
166             if(attacker>1) attacker=1;
167             if(NRP_network==0){
168                 ping_pending = true;
169                 out1.pingNRP(id,id, NRP_switch_id) after(5);
170                 ping_timed_out() after(ping_timeout);
171             }else{
172                 ping_pending = true;
173                 out2.pingNRP(id,id, NRP_switch_id) after(5);
174                 ping_timed_out() after(ping_timeout);
175             }
176             NRP_pending = true;
177             break;
178         case 2: //BACKUP :
179             heartbeats_missed_1++;
180             heartbeats_missed_2++;
181             if (heartbeats_missed_1 > max_missed_heartbeats && heartbeats_missed_2 >
                ↪  max_missed_heartbeats){
182                heartbeats_missed_1 =
                   ↪  (heartbeats_missed_1>max_missed_heartbeats+2)?max_missed_heartbeats+2:heartbeats_missed_1;
183                heartbeats_missed_2 =
                   ↪  (heartbeats_missed_2>max_missed_heartbeats+2)?max_missed_heartbeats+2:heartbeats_missed_2;
184                 // if(heartbeats_missed_1==heartbeats_missed_2 &&
                    ↪  heartbeats_missed_2==max_missed_heartbeats+1){
185                     // mode = PRIMARY;
186                     // heartbeats_missed_1 = 0; // Prevent detecting again immediately.
187                     // heartbeats_missed_2 = 0;
188                     // primary=id;
189                     // NRP_pending = true;
190                 // }else{
191                 if(NRP_network==0){
192                     ping_pending = true;
193                     //NRP_network=-1;
194                     out1.pingNRP(id,id, NRP_switch_id) after(15);
195                     ping_timed_out() after(ping_timeout);
196                 }else{
197                     ping_pending = true;
198                     //NRP_network=-1;
199                     out2.pingNRP(id,id, NRP_switch_id) after(15);
200                     ping_timed_out() after(ping_timeout);
201                 }
202                 NRP_pending = true;
203                 // }
204             }else if(heartbeats_missed_1 > max_missed_heartbeats|| heartbeats_missed_2 >
                ↪  max_missed_heartbeats){
205                 if(NRP_network==0 && heartbeats_missed_1 > max_missed_heartbeats) {
```

```
                    ping_pending = true;
                    out1.pingNRP(id,id, NRP_switch_id) after(5);
                    ping_timed_out() after(ping_timeout);
                }else if(NRP_network==1 && heartbeats_missed_2 > max_missed_heartbeats){
                    ping_pending = true;
                    out2.pingNRP(id,id, NRP_switch_id) after(5);
                    ping_timed_out() after(ping_timeout);
                }
                heartbeats_missed_1 =
                ↪  (heartbeats_missed_1>max_missed_heartbeats+2)?max_missed_heartbeats+2:heartbeats_missed_1;
                heartbeats_missed_2 =
                ↪  (heartbeats_missed_2>max_missed_heartbeats+2)?max_missed_heartbeats+2:heartbeats_missed_2;
            }
            break;
        case 3: //FAILED :
            break;
      }
      self.runMe() after(heartbeat_period);
    }
    msgsrv heartBeat(byte networkId, int senderid) {
        // if(?(true,false)) nodeFail();
        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 {
       Node nodeTarget1;
    }
    statevars {
        byte mynetworkId;
        int id;
        boolean which;
        boolean prevWhich;
        boolean failed;
        boolean amINRP;
        boolean primaryPinged;
        boolean terminal;
        Switch switchTarget1;
        Switch switchTarget2;
        int primary;
    }
    Switch (int myid, byte networkId, boolean endSwitch , Switch sw1, Switch sw2, int myFailTime) {
        mynetworkId = networkId;
        primary=0;
        id = myid;
        primaryPinged=false;
        terminal=endSwitch;
        amINRP = false;
        failed = false;
        switchTarget1 = sw1;
        switchTarget2 = sw2;
        which=true;
        if (myFailTime!=0) switchFail() after(myFailTime);
    }
    msgsrv switchFail(){
        failed = true;
```

```
276         amINRP=false;
277     }
278     msgsrv pingNRP_response(int senderNode,boolean w,boolean pw){
279         // if(?(true,false)) switchFail();
280         if(!failed)
281             if(terminal && senderNode <= MAX_SWITCHES) nodeTarget1.pingNRP_response(id, w,pw); //Pass back
282             else if(senderNode >id) switchTarget1.pingNRP_response(id, w,pw);
283         else switchTarget2.pingNRP_response(id, w,pw);
284     }
285     msgsrv pingNRP(int switchNode, int senderNode, int NRP) {
286         // if(?(true,false)) switchFail();
287         if(!failed)
288             if(terminal && NRP==id){
289                 prevWhich = which;
290                 which= (senderNode==primary);
291                 if(switchNode <= MAX_SWITCHES) switchTarget1.pingNRP_response(id,which, prevWhich);
                    ↪  //Response
292                 else nodeTarget1.pingNRP_response(id,which, prevWhich);
293             }else if(switchNode >id) switchTarget1.pingNRP(id,senderNode,NRP);
294         else switchTarget2.pingNRP(id,senderNode, NRP);
295     }
296     msgsrv new_NRP(int senderNode,int prim, int mNRP_network, int mNRP_switch_id) {
297         // if(?(true,false)) switchFail();
298         if(!failed){
299             if(id==mNRP_switch_id) {
300                 amINRP=true;
301                 primary=prim;
302             } else amINRP=false;
303             if(terminal && senderNode <= MAX_SWITCHES)nodeTarget1.new_NRP(id,prim, mNRP_network,
                ↪  mNRP_switch_id);
304             else if(senderNode >id) switchTarget1.new_NRP(id,prim, mNRP_network, mNRP_switch_id); //Pass
                ↪  back
305             else switchTarget2.new_NRP(id,prim, mNRP_network, mNRP_switch_id);
306         }
307     }
308     msgsrv new_NRPBack(int senderNode,int prim, int mNRP_network, int mNRP_switch_id) {
309         // if(?(true,false)) switchFail();
310         if(!failed){
311             if(id==mNRP_switch_id) {
312                 amINRP=true;
313                 primary=prim;
314             } else amINRP=false;
315             if(terminal && senderNode <= MAX_SWITCHES)nodeTarget1.new_NRPBack(id,prim, mNRP_network,
                ↪  mNRP_switch_id);
316             else if(senderNode >id) switchTarget1.new_NRPBack(id,prim, mNRP_network, mNRP_switch_id);
                ↪  //Pass back
317                 else switchTarget2.new_NRPBack(id,prim, mNRP_network, mNRP_switch_id);
318         }
319     }
320     msgsrv heartBeat(byte networkId, int senderNode) {
321         // if(?(true,false)) switchFail();
322         if(!failed)
323             if(terminal && senderNode <= MAX_SWITCHES) nodeTarget1.heartBeat(networkId,id)
                ↪  after(networkDelay);
324             else if(senderNode > id) switchTarget1.heartBeat(networkId,id) after(networkDelay);
325                 else switchTarget2.heartBeat(networkId,id) after(networkDelay);
326     }
327 }
328
329 main {
330     @Priority(1) Switch switchA1(DCN1):(1, 0, true , switchA2 , switchA2 , switchA1failtime);
331     @Priority(1) Switch switchA2(DCN1):(2 ,0, false , switchA1 , switchA3 , switchA1failtime);
332     @Priority(1) Switch switchA3(DCN2):(3, 0, true , switchA2 , switchA2 , switchA3failtime);
333     @Priority(1) Switch switchB1(DCN1):(4, 1, true , switchB2 , switchB2 , switchB1failtime);
334     @Priority(1) Switch switchB2(DCN1):(5, 1, false , switchB1 , switchB3 , switchB1failtime);
335     @Priority(1) Switch switchB3(DCN2):(6, 1, true , switchB2 , switchB2 , switchB3failtime);
336
337     @Priority(2) Node DCN1(switchA1, switchB1):(100, 100, 1, 4, node1failtime);
338     @Priority(2) Node DCN2(switchA3, switchB3):(101, 100, 3, 6, node2failtime);
339 }
```

D State Space

The state space of Timed Rebeca model for the NRP FD (including the problematic optimization) implementing case 7 of Table 1 has 70 states and 88 transitions. Case 7 is where switchA1 and switchB1 fail simultaneously at time 2500. A portion of the visualized state space is provided in Figure 8. We define the followings in the the property file (see L3):

```
DCN1Primary = (DCN1.mode ==1);
DCN2Primary = (DCN2.mode ==1);
DCN2Backup = (DCN2.mode ==2);
switchA1Failed = (switchA1.failed);
switchB1Failed = (switchB1.failed);
switchA1NRP = (DCN1.NRP_switch_id==1 && DCN2.NRP_switch_id==1);
...
```

The term $DCN1Primary$ means that the mode of DCN1 is PRIMARY (similar for DCN2) and the term $switchA1Failed$ means that the state variable $failed$ of switcheA1 is $true$ (similar for switcheB1). $switchA1NRP$ means that the state variable NRP_switch_id equals 1 (the id of switchA1) for both DCNs. In case 7 of Table 1, both switches fail at time 2500. As we are at the time 3000 in S59, switchA1Failed and switchB1Failed are true at the states depicted. Both DCN1 and DCN2 execute a runMe in each heartbeat period:

```
heartbeat_period = 1000 // line 1 of Listing 1
...
self.runMe() after(heartbeat_period) // line 35 of Listing 1
..
```

In each period, PRIMARY (DCN1) checks its NRP availability. In the state S63, DCN1 sends a PINGNRP message to switchA1 in the new heartbeat period, @3000. By receiving PINGNRP, switchA1 which is failed, does nothing (line 296 of Appendix C). In the state S65, by running Ping_timed_out, DCN1 will notice that switchA1 has failed. DCN1 tries to select a new NRP from its NRP candidate set (here switchB1 which is not operational at the moment). Note that there is no active NRP in S66. At the next runMe, @4000, DCN2 changes its mode to PRIMARY due to missing more than maximum heartbeats allowed on both networks simultaneously. We can see in S70 a dual primary situation occurred. We commented out the assertion such that the model checker continues creating the state space.

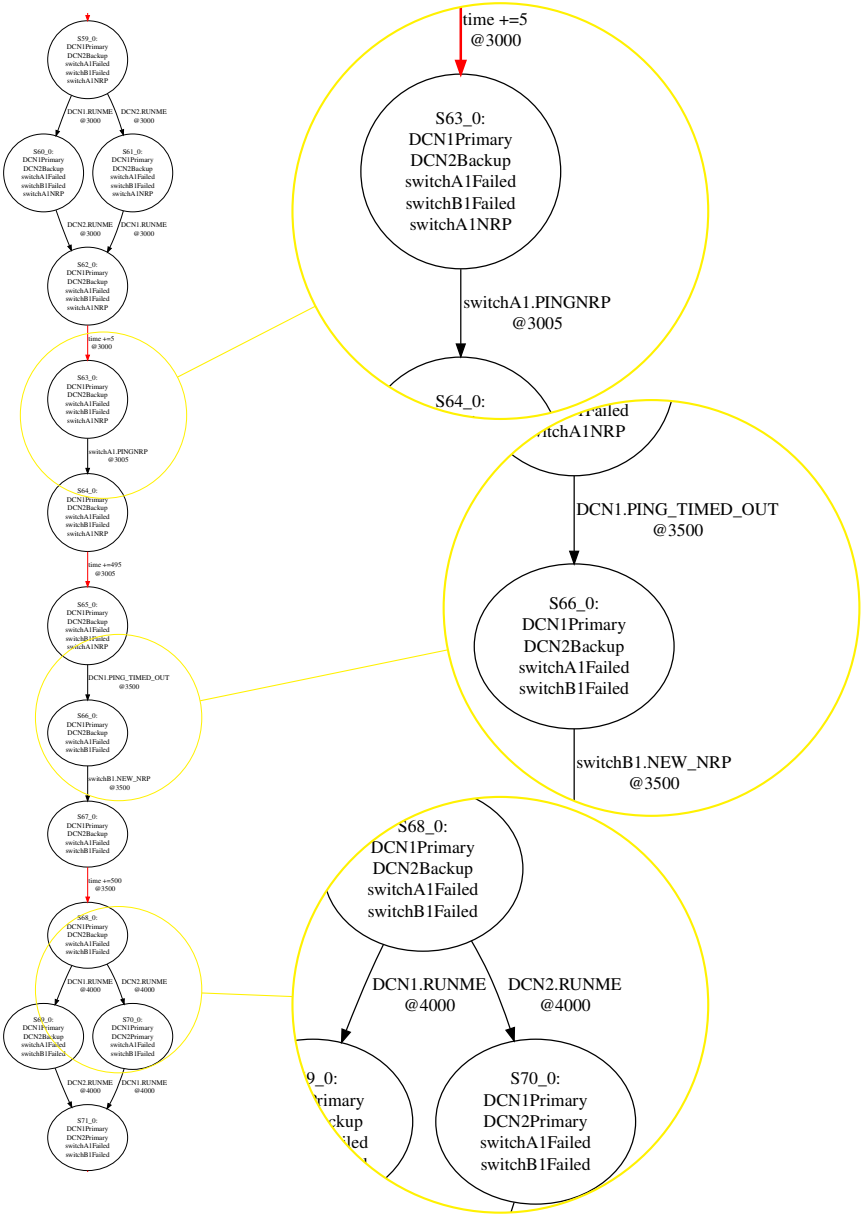


Figure 8: A part of the visualized state space for the Timed Rebeca model of the NRP FD.