Modeling & Validation of a Fault-Tolerance Protocol

Marco Zoveralli, *Master Student, EPFL*

***Abstract*—Fault-Tolerance protocols are very often complex systems and their implementations do not always have a formal proof for correctness. In this project, we want to validate an internally developed fault-tolerance protocol: Axo. We mainly focus on two components: the detector and the rebooter. The requirements and an implementation of these are available. We firstly show how these two components can be modelled. Then, we show how we can express the requirements of the system as properties, and how we can formally verify them with the support of proper tools. We also show how this activity –modeling the system– was repeated multiple times in order to improve the models and to be able to verify more meaningful properties. Finally, we show the main properties that have been verified and that characterize the fundamental behavior of the detector and the rebooter.**

I. Introduction

FAULT-TOLERANCE protocols for real-time systems usually consist in complex distributed algorithms, such as when it comes to tolerating delay faults. This is the case of Axo 9, a protocol developed at the Laboratory for Communication and Applications at EPFL. In particular, Axo has been designed to tolerate delay faults in real time systems. These systems imply very often working in safety critical environments with real-time mission critical control applications. Mission Critical Systems are systems that play a key role in the main activities of an organization: without them, the whole functioning may be compromised. In some cases, these activities have a huge impact not only on the organization that provide the service, but also to the people that are involved. Some very relevant examples of these kinds of systems can be: an electric power systems, a railway, an aircraft navigational system or an online banking system. In this context, any kind of fault can lead to disastrous consequences, that can span from very high cost losses for the business to life loss. This is the reason for which it is imperative to have implementations that are as correct as possible, in order to avoid faults and, in the worst case scenario, to properly handle them. Correctness is a broad concept, and there are multiple ways to approach it. In this project, we focus on approaching correctness through **model checking**: the main goal is to model the aforementioned protocol. Then, we want to check that it satisfies the defined requirements. These requirements have to be expressed in form of properties, which can be formally checked with the support of proper tools, called model checkers. Verifying properties, depending on the assumptions that have been made, allows to check the specific requirements. The higher the number of these properties, the higher the strongness of the model. The existing Axo implementations suffer from the fact that they are based on algorithms that came with no formal proof: there is no guarantee that these implementations will behave in the way they are suppose to behave. In fact, test cases that can be provided are limited and they are not enough to prove that an implementation is really correct. Having a complete model means having a reference on which future, improved, Axo implementations may be based.

A. Report Structure

This report is organized in the following manner:
In section II, an overview of the proposed solution will be given, pointing out the main steps and decisions that have been taken.
Section III illustrates the details of the proposed solution, quantifying the various results that have been achieved. In particular, the modeling and verification phases will both be detailed for the sake of this project, and the taken choices will be motivated. Furthermore, the unexpected difficulties that have been met will also be described.
Section IV briefly describes the skills that were used during this project and the skills that have been acquired.
Section V concludes this report, by providing self-assessment and by giving some insights for possibile future work.
The described models are all available on the EPFL github repository [1].

II. Solution

THE solution for this problem consisted in going through the phases of model checking (see appendix D) repeatedly, starting from simple models and gradually increasing their complexity in order to verify more intesting properties. The goal of this section is to give an overview of the proposed solution. Details can be found in the successive sections.

A. Tools and Languages

Tools and languages play a key role in model checking. A satisfactory language is one that allows to easily represent finite state machines of non-trivial complexity. Also, it should allow to realize models that are compatible with model checkers. Finally, if it also allows to generate some executable code, it is even better.
In this project, BIP 11 modeling language was chosen, due to its proper expressional power in terms of finite state machines and the possibility to model complex systems in a relatively easy way. This language lacked of a system for performing verification: this is why we relied on external tools. One

[1] https://git.epfl.ch/polyrepo/private/files.go?repositoryId=11051&branch=BIPMarco

useful tool for checking safety properties is **BIP Checker**[2]. However, we preferred to use something that would allow to express properties through LTL/CTL syntax. The theorem prover **NuXmv**[3] was chosen. Due to the characteristics of BIP and NuXmv, modelling was always performed with the former, while verification was achieved through the latter. We also want to point out that BIP allows to generate C++ code directly from the model, while NuXmv does not. This is another reason that makes BIP a proper tool for building models. Since NuXmv uses a different language for modeling finite state machines, we also used a translation tool, to convert BIP models to NuXmv models: **BIP-to-NuSMV**[4]. This led to some consequence, as shown in III-L.

B. Algorithms for Model Checking

There are several algorithms that are used for model checking. Over the past three decades, many techniques have been developed and a lot of results about which one provides best performance were given. Several papers exist about this matter. One of the most important results is the Kennet L. McMillan's "Approach to the state explosion problem": **Symbolic Model Checking** 1. It can be defined as a pillar of model checking, since that approach was the basis for a countless number of techniques that were developed during the following years. Symbolic model checking avoids building a state graph to traverse the state space, by using boolean formulas to represent sets and relations. It was firstly implemented through BDD (Binary Decision Diagram) 1, which implied expressing properties through a CTL syntax. After further research, symbolic model checking was implemented through SAT (Propositional Satisfiability Problem) 2, which implied expressing properties through a LTL syntax. Several techniques rely on SAT, including BMC (Bounded Model Checking) and IC3 [4, 5, 6, 7, 8].
There are several discordant opinions about which family of techniques is better, among BDD-based and SAT-based methodologies 3. Some major results state that these two large families are complementary 4 and therefore it depends on specific cases.
In this project, our approach was to define properties through CTL syntax. If the state explosion problem would occur for some statement, then we would express the same property in an LTL-equivalent one and test its verification through the SAT-based algorithms. Many times, SAT-based techniques have revealed themselves to mitigate the state explosion problem better than BDD-based ones, especially for what concerns safety properties. The reason is that they occupy much less space 2.

C. Modeling

Axo is a large protocol A. We mainly focus on two components: **Detector** and **Rebooter**. The reason for this choice is that these components implement fundamental characteristics of Axo, their requirements are important for the functioning of the protocol and their behavior is complex. This is less true for other elements, such as the Masker, which is more vital but it is simpler and a formal proof of its correctness is already given.
When it comes to model a system, there are mainly two (opposite) approaches: **refinement** and **abstraction**. As a model is more refined, the number of properties that can be verified is higher too. This is why, generally speaking, refinement is preferable over abstraction. However, too large models are not easily manageable by model checkers, and therefore a tradeoff is necessary: broadness of properties against the time needed to verify them. For both the detector and the rebooter, different versions have been realized, starting from the simplest – most abstract version, with several assumptions– to the most complex –more refined version, with less assumptions– . Also, the components were considered as separate models at first and they were put together in a second time. One challenge, during the design, was posed by the fact that modeling languages are declarative ones: many activities that could be easily done with any object oriented programming, here require deep reasoning before being implemented. For instance, the BIP modeling language requires that each component is explicitly declared and initialized. Therefore, no data abstractions –even simple ones such as arrays– are allowed. This implies, for example, that in each model a fixed number of components is defined.

Data transfers, in BIP, occur through **connectors**. Connectors involve multiple atoms and one important design decision, that characterized the whole work, was based on the fact that the BIP models would scale on connectors rather than atoms: this allowed to reduce the duplicate code –which, however, was unavoidable. Also, scaling over the connectors meant being able to isolate the changes of any components and to add new features. However, the most important factor that derived from this decision was the reduction of ports: this design choice caused the number of ports not to increase too much and therefore the complexity of the model would grow slower. This principle guided the design of all the models. In fact, modeling has a direct impact on any kind of problem that occurs during the verification of properties: reducing the number of possible states is the first measure that helps in mitigating already mentioned state explosion problem. The number of states of a model grows exponentially. In order to avoid confusion, it is recommended to understand the difference among the "state of the model" and the "state of the atom" II-E.
Indeed, another factor to decrease the number of states is the number of bits representing the variables: a variable of n bits will add a multiplicative factor of 2^n to the number of total states of the model. This issue was experienced in the super refined model (see Appendix C) of the detector: this is why all the other models encode the database values through representative ones (very few bits used for these variables, which implies having an extremely reduced number of values).
One last note: every model that is shown in the following sections is lacking of the variables, which are a fundamental part of the state. However, the whole code that implement those models is available on the git repository.

[2]http://risd.epfl.ch/bipchecker
[3]https://nuxmv.fbk.eu/
[4]http://risd.epfl.ch/bip2nusmv

D. Verification

Verification was performed on each version of the components. Once some abstractions were ready, the first approach was to begin with very simple statements, while completely ignoring the assumptions that were made. Then, subsequently, some of these assumptions were removed and more properties were verified according to the same methodology. At each step, we also made sure that old properties would still hold, although some assumptions were removed. At the end, detector and rebooter were merged in a single model and one goal was to check whether they still behaved correctly and that they could properly interact among each other.
It is worth to point out that during each phase we tried to minimize the impact of any future change that would be necessary on the model. However, due to the nature of modelling languages –declarative ones– this was not easy and not always possible. The next section will give more details about the proposed solution, in terms of quality and quantity of the work that has been done, including the unexpected difficulties that have been met.
In order to proceed with verification, whenever a BIP model was ready, it had to be converted to a NuXmv model. This required another intermediate phase, which implied converting the BIP models in the old syntax [5]. In order to check the properties for all the possible values of the variables, NuXmv models require these variables to be non-determistically chosen (since their values are often handled by components that are assumed to work well).
Useful information can be found in appendix B.

E. State of the model VS state of the atom

It should be clear that ”state of the model” and ”state of the atom” are two extremely different things. In fact, the state of an atom is simply a variable. Each atom has, mainly, three kinds of variables: states, ports and ”classical” variables. Each combination of these variables, considering all the atoms, defines a unique state of the model. Since ports are boolean variables, each port doubles the number of states in a model. This is why their number should be kept low. Generally speaking, ports of an atom can be used multiple times for different transitions, since the state of the model is identified also by the state of the atom itself.
As an example, we consider a simple model with three detectors, each one represented by an atom. For the sake of simplicity, we assume that they have their own variables and transitions and that they are not bounded each other. Each detector (atom) has a set of possible states (also called **places**), which can be considered as a variable itself. The same holds for the transitions: they are called ports but they can be considered as binary variables.
In this model, each detector's state is defined by the **place** variable, while the state of the whole model is defined by all the variables at any given time (which includes that states of the atoms). This distinction plays a key role when properties have to be verified.

[5] http://risd.epfl.ch/bip2nusmv

III. Achievements & Major Events

MOST of the work of this project relates to modeling the detector and the rebooter, two of the main components of Axo. The following subsections contain detailed descriptions of these phases, along with the design choices that were taken and the issues that were caused/avoided by them. All the implementations of the subsequent models are stored on the github repository. For each model, the BIP and NuXmv versions are available, along with a file that contains the verified properties (sometimes with a little comment). In some abstractions, the properties are also included in the NuXmv models. Check Appendix B for more information on how to quickly verify them and on how to understand them B-A.

A. Detector

The detector is the component responsible for detecting crash and delay faults. It receives validity reports from the masker, and uses them to detect faults in replicas (including itself). To do this, it maintains a database with record for each replica. The validity reports contain the timestamp of the setpoint, the ID of the replica that issued the setpoint, the health of the replica that issued the setpoint, the detector timestamp (computed as the highest setpoint timestamp that the detector of the issuing replica had processed), and a flag that shows whether the setpoint received at the masker was valid. Each record of the database contains the highest received setpoint timestamp, the highest received detector timestamp, the replicas health as seen by this replica and a flag showing whether the last sent setpoint was resulted in a delay-fault. The detector was the first component to be modelled. Each model has its own characteristics, but there are some common features. One common assumption is that all the other components behave well. All the components of the model have an ID that is coherent with their name. For instance, detector1 will have an ID that is 1. The models have been realized with BIP, which is able to represent the described finite state machines (models) in a relatively simple way. More information in appendix E and in 10.
Requirements: we need completeness and accuracy. Therefore, the detector must be able to detect faults of replicas. Furthermore, if a replica is detected to have crashed, then a fault has occurred. These requirements were translated in a set of properties that depended on the level of refinement of each model.
The main two operations that are performed by the detector are the update of the database and the checks for delay/crash faults.
Before looking at the properties, appendix B-A might be helpful.

B. Detector: Super-refined model

Although the implementation of this model required a lot of time, it really did not provide any useful result in terms of model checking. For this reason, its design is still included, but only in Appendix C. On the other hand, this model can be still useful as a point of start for other models. We also remind that BIP models can be used to generate executables.

C. Detector: Model 1

Assumptions: several. The goal of this model was to get started with model checking.

1) Communications: they are assumed to be perfectlty synchronized.

2) Handled data: everything is encoded with symbolic values that are different from the real ones. Timestamps, health values, identifiers, among others, are the handled by the database as dummy values and their purpose is to trigger the right events. This allowed to decrease the number of possible states dramatically, since NuXmv generates states that have all the possible values that are assumed by the various variables (as mentioned in II-C).

3) Number of components: Three detectors and one dummy masker.

4) Operations: we assume that the detectors know how to perform the operations (update of the database and fault detection/report) and when to perform them. Therefore, it can already recognize old/new validity reports and it reports crashes if and only if they occur.

Design: the design was quite simple, since we simply wanted to check whether the detector would eventually be ready to process any validity report that it would receive. Basically, for each detector, there is one atom that represents the detector itself and one representing the database. These two atoms contain very similar states, but the latter has some states that refer to the specific record on which the operation has to be performed. Each replica is represented by a record of the database (on each detector). Therefore, when an update has to be performed, the detector should choose which record has to be updated. This is done on the database and it is represented through the states CHOOSE_RECORD1, CHOOSE_RECORD2 and CHOOSE_RECORD3. On the detector atom, there is only one state of this kind: CHOOSE_RECORD. In this way, it is possible to separate the updates (UPDATE1, UPDATE2 and UPDATE3) that relate to different replicas. From the detector perspective, it is not important to know exactly what happens in the database. To represent the checks, there are the respective ”symbolic” states that do not really trigger any operation, since we are assuming that the detection is handled well.

Verification: since the major operations –database's update and the faults' checks– are assumed to be performed well, the main checked properties simply indicate whether the these operations are eventually performed and that a detector always processes the received reports and that it is ready to receive new ones.

- One property: SPEC AG (dr1.Nuplace = NuSREPORT_RECEIVED ->AF (dr1.Nuplace = NuSUPDATED)). It means that whenever the detector receives a report, it eventually performs the update of the database. We verified similar properties that involved other states as well and the other detectors.
- Another property: SPEC AG (dr1.Nuplace = NuSREPORT_RECEIVED ->((AX dr1.Nuplace = NuSREPORT_RECEIVED) |(AX dr1.Nuplace = NuSCHOSEN_RECORD))). It means that whenever the detector receives a report, its next state (of the atom) will be the one in which it chooses the record, of the database, to update. This has been verified for the other states as well. By doing this for all states, we can deduce a set of properties that is stronger than the liveness property that we expressed above. And in many cases it takes less time to verify this set of properties, because they are safety. See the .smv and properties files of this model for more details. Clearly, with this set of properties we automatically verify that a detector is always ready to process new validity reports.

D. Detector: Model 2

Assumptions:

1) Communications: same as model 1.

2) Handled data: same as model 1.

3) Number of components: same as model 1.

4) Operations: we still assume that the detectors know how to perform the operations (update of the database and fault detection/report). However, while we are still making assumptions on the detection of faults, we do not make any assumption about the validity report's processing.

Design: the design is very similar to the previous model. However, in addition to UPDATE1, UPDATE2, UPDATE3, some states were added in order to represent whether the database was updated at the right time or not: NOUPDATE1, NOUPDATE2 and NOUPDATE3. This is shown in figure 1.

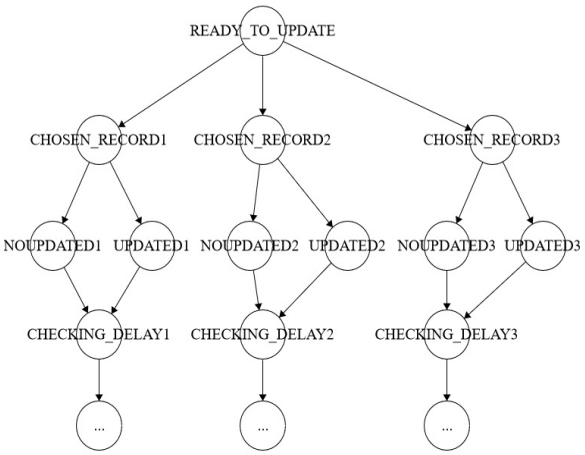

Fig. 1. Database Update

For this purpose, the controller timestamp field of the received validity report was also used. As already stated, timestamps are encoded through simple values. In this case, a binary value was chosen: if the controller timestamp is 0, then it means that the report is old and therefore no update should occur. If the controller timestamp is 1, then it means that the report is new and therefore an update should occur. In either case, the detector performs faults checks immediately after.

Verification: we can check that the database is updated only when the validity report is new.

- Property 1: SPEC AG (dr1.Nuplace = NuSUPDATED ->dr1.NuVcontrollerTimestampValidityReport =

m1.NuVone). It guarantees that if a detector has performed an update, then the controller timestamp of the received validity report is 1.

- Property 2: SPEC AG (dr1.Nuplace = NuSCHOSEN_RECORD & dr1.NuVcontrollerTimestampValidityReport = m1.NuVone ->(AX (dr1.Nuplace = NuSCHOSEN_RECORD) |AX (dr1.Nuplace = NuSUPDATED))). It means that, whenever a detector has to decide whether to perform an update or not and it has received a new validity report, it will perform the update in its next state (state of the atom).
- Similar properties can be verified when the controller timestamp in the validity report is 0.

The old properties (the ones of model 1) still hold.

E. Detector: Model 3

Assumptions:

1) Communications: same as model 1 and model 2.

2) Handled data: same as model 1 and model 2.

3) Number of components: One replica. The reason of this choice was that detectors do not really communicate each other, and therefore lightening the model in this way would make it possible to refine the model more, without having its size to grow too much.

4) Operations: we still assume that the detectors know how to perform the operations (update of the database and fault report). However, no assumption is made on the fault detection's phase.

Design: we added some states in order to represent the detection of a fault, after the respective check. It is important to notice that these states are added on the database atom, and not on the detector. In fact, from the detector's atom perspective, everything is almost like before: the checking phases are simply represented through a checking state. This is one benefit that we had by scaling over connectors: we isolated the change of the database. There is a state that indicate whether that replica is faulty or not. Here we are representing the delay faults and crash faults.

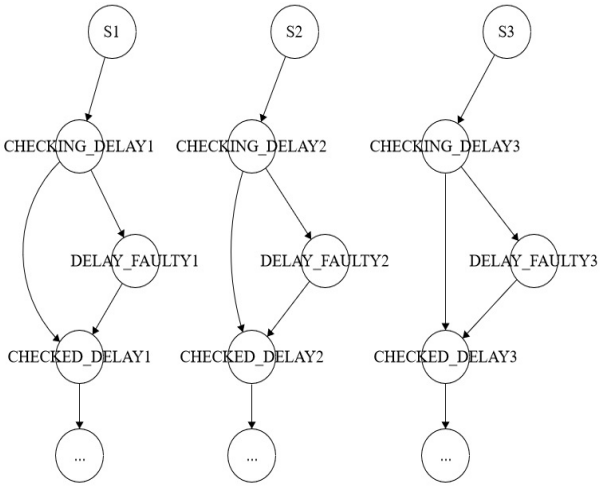

Fig. 2. Checking Delay Faults

As shown in figures 2 3, the added states are: DELAY_FAULTY1, DELAY_FAULTY2, DELAY_FAULTY3, CRASH_FAULTY1, CRASH_FAULTY2 and CRASH_FAULTY3. In order for a replica to be detected as delay faulty, it means that the health is below a defined threshold. This can also be interpreted as a certain number of consecutive faulty validity reports received (since the health is decreased each time a faulty validity report is received). Therefore, each replica has a variable that represents the number of consecutive faulty reports that are received. This variable is incremented if a faulty validity report is received. Otherwise, it is decreased. When this variable reaches the maximum value, then the related replica is detected as faulty and the counter is reset. Image 2 gives an idea of how this has been implemented. Notice that S1, S2, S3 are states of the same atom and it simply relates to the replica for which the report has been received. S1 could be either UPDATED1 or NOUPDATED1. The same holds for S2 and S3.
Similarly, figure 3 shows our design decision for what concerns the detection of crash faults. Crash fault's checking requires determining the maximum timestamp and comparing with the timestamp that relates to the checked replica (i.e. the replica indicated by the identifier in the validity report).

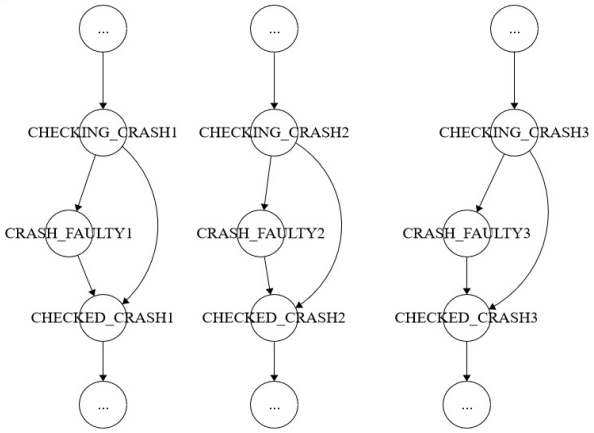

Fig. 3. Checking Crash Faults

If the timestamp of the checked replica is "too much" smaller than the maximum timestamp (a fixed T defines the size of this "too much". This T represents a timeout), then the replica is considered as crash-faulty. In our model, this process is simplified, since we do not handle timestamps. Therefore, there are four possible values that can be assumed by the maximum timestamp: 0, 1, 2 and 3. 0 means that the maximum timestamp is such that no replica has an associated timestamp that is too much smaller than this maximum timestamp. In this case, no replica would be detected as crash-fauly. 1 means that the maximum timestamp is such that it is much greater than the timestamp of the replica 1. In this case, replica 1 would be detected as crash-faulty. 2 and 3 have a similar meaning of 1, but they relate to replica 2 and replica 3, respectively. Although this simplifies the model, it imposes a limit: we do not know the relation between the replica's timestamps. This means that we do not know whether one timestamp is bigger

or smaller than another one. Therefore, a replica may be not considered crash-faulty even if it should. We are considering only the timestamps of the controllers for the crash faults, since the timestamps of the detectors have a similar role and it would make sense to use them in more refined models.

Verification: The aforementioned states would play a key role for both completeness and accuracy verification.

- Property 1: SPEC AG (db1.Nuplace = NuSDELAY_FAULTY1 ->db1.NuVconsecutiveFaultyReports1 = m1.NuVmaxConsecutiveFaultyReports). This property ensures that if the replica with ID 1 is detected as faulty, then the number of received consecutive faulty reports is such that the health is below the defined threshold.
- Property 2: SPEC AG ((db1.NuVconsecutiveFaultyReports1 = m1.NuVmaxConsecutiveFaultyReports ->(AF db1.Nuplace = NuSDELAY_FAULTY1))). It means that if the maximum number of faulty reports, that relate to the replica with ID 1, have been received consecutively, then the replica is eventually detected as delay-faulty.
- Property 3: SPEC AG ((db1.NuVconsecutiveFaultyReports1 = m1.NuVmaxConsecutiveFaultyReports & db1.Nuplace = NuSCHECKING_DELAY1) ->(AX db1.Nuplace = NuSDELAY_FAULTY1 |AX db1.Nuplace = NuSCHECKING_DELAY1)). This is stricter than the previous one: it says that if a detector is performing a check on delay faults, and if it has received the maximum number of faulty reports from replica 1, consecutively, then the next state of the atom will trigger this fault. Notice that this is faster to prove than property 2.
- Property 4: SPEC AG (db1.Nuplace = NuSCRASH_FAULTY1 ->db1.NuVmaxControllerTS = 0ud4_1). This property ensures that if a replica is detected as crash-faulty, then the maximum controller timestamp much greater than the timestamp related to replica 1 ("much" is quantified by the timeout).
- Property 5: SPEC AG (db1.NuVmaxControllerTS = 0ud4_1 & db1.Nuplace = NuSCHECKING_CRASH1 ->(AF db1.Nuplace = NuSCRASH_FAULTY1)). The meaning of this property is that if a detector is performing a check for crash faults and the maximum controller timestamp is much greater timestamp related to replica 1, then the replica is eventually detected as crash-faulty.
- Property 6: SPEC AG (db1.NuVmaxControllerTS = 0ud4_1 & db1.Nuplace = NuSCHECKING_CRASH1 ->(AX db1.Nuplace = NuSCRASH_FAULTY1 |AX db1.Nuplace = NuSCHECKING_CRASH1)). This is similar to the previous one, but it is a safety property. Like property 3, this is stronger than property 5 and it is also faster to check with the model checker.

These properties have been verified also for the other replicas in the database. The properties of the previous models still hold.

F. Detector: Model 4

Assumptions: the same as model 3, but the assumption of the perfect synchronous communication was removed. A buffered synchronous channel was realized.

Design: The buffer is represented by an atom which plays an intermediary role between the masker and the detector: the masker may send validity reports repeatedly before the detectors finish to process them. The buffer would store these messages and the detectors can read them in the right order (from the first received to the last one) from the buffer. The buffer has a fixed size (3, in our case) and its state is represented by the number of messages that have been received and that have to be read. Each state allows either to read (unless the buffer is empty) and write (unless the buffer is full). Also, the order of messages is important. The design is shown in figure 4. It can be considered as an implementation of a circular array [6], in which each state is represented by two indexes: one indicates the place of the array on which we will write the next data, the other indicates the place of the array from which the next data has to be read. Whenever a write (read) occurs, the first (second) index is increased. Notice that a state can have two equal indexes in two cases: when the buffer is full (only read operations allowed) and when the buffer is empty (only write operations allowed).

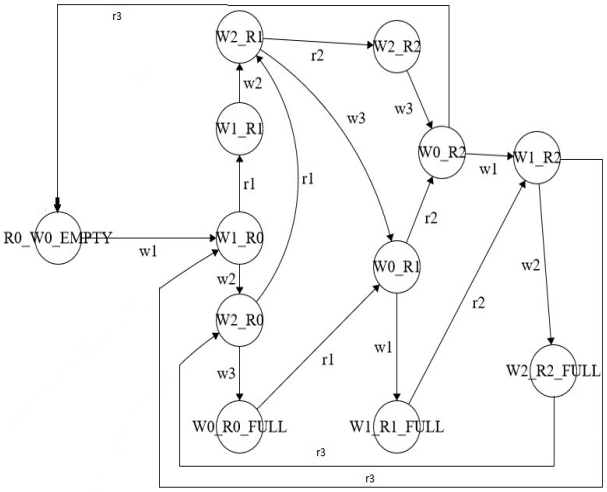

Fig. 4. Buffer

This is why there are some states that seem to be redundant. The complexity of this data structure (number of states) is $n^2 + n$. The data handled by the buffer, in this model, are all the data that compose a validity report.

Verification: Although this design allows to implement a more realistic model, the complexity brought by this component is not negligible: several properties are harder to express and, above all, very often they cannot be verified when expressed through CTL syntax, because it takes too many resources in terms of time and memory. Therefore, most of the properties could not be verified through CTL syntax. We used LTL syntax + SAT-based algorithms and the outcome

[6]https://en.wikipedia.org/wiki/Circular_buffer

was successful in most of the cases, for what concerns the main properties of the model.

G. Rebooter

The rebooter is the component that is responsible for rebooting the replicas of the controller. Whenever the detector detects a fault (crash or delay) it sends a detection message to its local rebooter. If the replica that has to be rebooted is the local one, the local rebooter will take care of that. If it is another one, the rebooter will send a recovery message to the rebooter of that replica, which will handle the recovery process.
Again, the models have different characteristics. All the components of the model have an ID that is coherent with their name. For instance, rebooter1 will have an ID that is 1. The models have been realized with BIP, which is able to represent the described finite state machines (models) in a relatively simple way. Some common features are described in appendix F and in 10. Note that we do not make any distinction among crash faults and delay faults: the rebooter simply sees a detection message from the detector.
Requirements: a rebooter must reboot the replica when it is detected as faulty. Also, when a detector identifies a fault of a replica, this replica will be eventually rebooted. Recovery messages are categorized in internal and external, and they depend on the received detection messages. These requirements were translated in a set of properties, that depended on the refinement level of the implemented models.
Before looking at the properties, appendix B-A might be helpful.

H. Rebooter: Model 1

Assumptions: in this model, the main assumption relates to the fact that the detector was considered as a functioning component. The reason behind this choice is that several other abstractions have been used to verify properties of the detector and therefore we preferred to keep this model as simple as possible. Another strong assumption is that the rebooters handle external recovery messages and correctly reboot the involved replica.

1) Communications: they are assumed to be perfectlty synchronized.

2) Handled data: the main data is represented by the replica ID of the detectors and the rebooter and the ID of the faulty replica.

3) Number of components: two (dummy) detectors and two rebooters.

4) Operations: we assume that the rebooter can perform the reboot of replicas correctly, once the correct recovery process has been triggered (internal or external).

Design: We put some states that indicate the step of the recovery process. Thus, the two main states are for the kind of recovery process that is triggered: INT_REC_BEGIN and EXT_REC_BEGIN (figure 5).

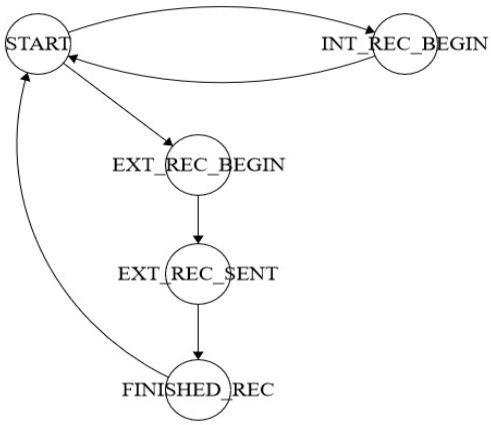

Fig. 5. Rebooter - Model 1

In fact, based on the detection message that is received from the detector, the rebooter will either begin an internal recovery process or an external one. In the first one, the rebooter will simply reboot the replica. In the second one, it will communicate with the other rebooter, which will reboot the replica. This latter part is not modeled in this abstraction.

Verification: the properties that were verified mainly check whether the rebooter goes through its states meaningfully, which means that the the detection performed by the detector should trigger the right recovery process.

- Property 1: SPEC AG (r1.Nuplace = NuSINT_REC_BEGIN ->r1.NuVfaultyReplicaID = 0ud3_1) . It means that if a rebooter starts an internal recovery process, then the detection message that it has received is reporting a fault on its replica ID.
- Property 2: SPEC AG (d1.Nuplace = NuSSENT_INTERNAL ->(AX r1.NuVfaultyReplicaID = 0ud3_1)). It means that if a detector sends an internal recovery message, then its local rebooter will see a detection message that is reporting the fault on the replica with the same ID.
- Property 3: SPEC AG (d1.Nuplace = NuSSENT_INTERNAL ->(AX r1.Nuplace = NuSSTART |AX r1.Nuplace = NuSINT_REC_BEGIN)). Can be deduced from the previous ones.
- Property 4: SPEC AG (r1.Nuplace = NuSSTART & d1.NuVfaultyReplicaID = r1.NuVreplicaID ->((AX r1.Nuplace = NuSSTART) |(AX r1.Nuplace = NuSINT_REC_BEGIN))). Similar to the previous one.

All these properties guarantee that an internal recovery process is triggered only when a detector is sending an internal detection message. The same holds for the external recovery process, and similar properties have been defined to verify this. They mostly relate to the rebooter with ID 1, but it is easily extended to the other rebooter as well. In addition to these properties, like for the detector, we checked some statements that related to the fact the states are all traversed when they are supposed to.

I. Rebooter: Model 2

Assumptions: Very similar to the previous ones. However, here the external recovery messages are automatically handled only partially. In particular, we still assume that the rebooter correctly reboots the replica, but these operations are represented in the model. The following paragraph explains this. Last, we assumed to use a synchronized buffered channel.

1) Communications: we assume a synchronized buffer channel, like for the model 4 of the detector (4). Data handled by the buffer: ID of the faulty replicas.

2) Handled data: same as model 1.

3) Number of components: same as model 1.

4) Operations: we assume that the rebooter can perform the reboot of replicas correctly, once the correct recovery process has been triggered (internal or external). However, we did not assume that the external recovery process is automatically handled.

Design: the main changes, in comparison with the previous model, are that two new atoms have been added: one for the buffer and one for handling the external recovery messages. Due to the presence of the buffer, we decided to put an additional state to the rebooter, signaling only that a detection message has been received (RECEIVED_DETECTION_MESSAGE). An additional transition is used by the rebooter, in order to determine whether the detection message should trigger an internal recovery process or an external one. The atom that handles external recovery messages acts like a thread of the rebooter: it simply handles the message while the rebooter itself may be doing other things. This is far more realistic than the previous one.

Verification: unlike the detector, here the addition of the buffer does not slow down too much the verification of properties. This is mainly due to the size of this model, which is noticeably smaller than the detector's. It is important to notice that all the properties that do not directly involve the detector are still verified. This is a significant result, since the introduction of a buffer may have caused troubles –like in the case of the detector, when, due to the model's size, everything became harder to check. When the properties include the detector as well, obviously there has to be a change in their definition. In fact, since there are independent atoms in this model, assuming that something involving multiple atoms eventually happens is simply wrong. For instance, there are two rebooters. If we know that there is the detection of a fault on a detector, we cannot say anything about the immediate next states of the model: we need to include the buffer in our properties. We cannot even say anything about liveness properties of a specific rebooter, because, since there are independent atoms, the state of the model could simply change in relation to what occurs on the other rebooter, for an undefined time (notice that this is true for the previous model as well). However, we can still write some interesting statement, if we consider the buffer.

- Property 1: SPEC AG (d1.NuVfaultyReplicaID = 0ud3_1 & d1.Nuplace = NuSSTART ->((AX d1.Nuplace = NuSSTART) |(AX (buffer1.NuVfaultyReplicaID1 = 0ud3_1 |buffer1.NuVfaultyReplicaID2 = 0ud3_1 |buffer1.NuVfaultyReplicaID3 = 0ud3_1)))). This property ensures that if the detector has detected an internal fault, then during its next state it will send this detection message to the buffer. This property also shows that the next one is non-trivial.
- Property 2: SPEC AG (((buffer1.NuPread1 = TRUE & buffer1.NuVfaultyReplicaID1 = 0ud3_1)) & r1.Nuplace = NuSSTART ->AX (AX (r1.Nuplace = NuSRECEIVED_DETECTION_MESSAGE |r1.Nuplace = NuSSTART |r1.Nuplace = NuSINT_REC_BEGIN))). This property says that whenever the buffer has a detection message that relates to the replica with ID 1, and if that message is the next one to be read, then the internal recovery process will be started on the rebooter. This property does not include the detector. But, clearly, the message is coming from the detector. Inside the .smv file of the model, this property is extended with the other places of the buffer too: they were not included inside this report in order to save space.

The other properties are very similar to the ones that were verified on the detector: they ensure that each state triggers the right successive state. For instance, if an external recovery process starts, we also need to verify that the handler (of the other rebooter, since it is external) receives the message.

J. Rebooter: Model 3

Assumptions:

1) Communications: same as model 2.

2) Handled data: in addition to model 2, we also handle the timestamp at which the fault was detected.

3) Number of components: same as model 2.

4) Operations: We did not assume anything about the decision process, neither internal nor external. The rest is like the previous models.

Design: new states and ports have been added, in order to represent the decision process of the rebooting phase. In fact, a replica should be rebooted only if the timestamp has a particular value: it has to be higher than the last reboot timestamp. A timeout is also considered, in order to avoid to reboot a replica too quickly. This is encoded through a binary variable that triggers the decision process (0 = reboot, 1 = do not reboot). By looking at figure 6, we can notice some differences. First, there is an additional intermediate state: START_REBOOT. Then, there are two states indicating whether, after an internal detection, the replica has been rebooted or not. This decision is taken according to the value of the detection timestamp. The rest is quite similar to the previous model. The external handler now handles the recovery messages in the same way as the internal one does. The design of the handler is simple and it is understandable directly from the BIP code.

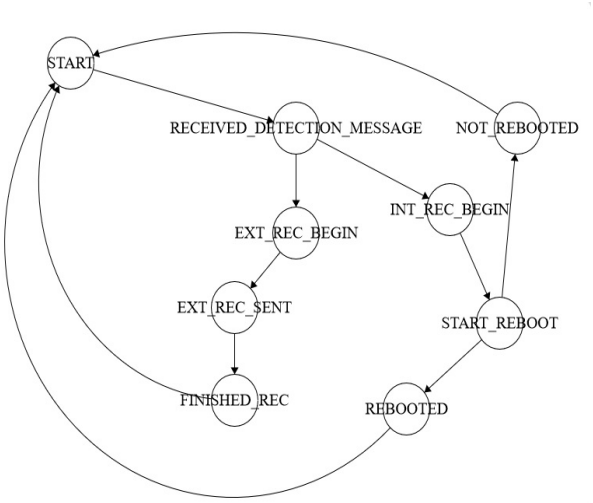

Fig. 6. Rebooter - Model 3

Verification: In order to verify the properties of this model, we used LTL syntax instead of CTL, in order to mitigate the state explosion problem. In particular, by using an algorithm that is based on K-liveness 6, we managed to verify the properties in this model too. Here we have the properties of the previous model, but converted to an LTL syntax.

- Property 1: LTLSPEC G (d1.NuVfaultyReplicaID = 0ud3_1 & d1.Nuplace = NuSSTART ->((X d1.Nuplace = NuSSTART) |(X (buffer1.NuVfaultyReplicaID1 = 0ud3_1 |buffer1.NuVfaultyReplicaID2 = 0ud3_1 |buffer1.NuVfaultyReplicaID3 = 0ud3_1))))
- Property 2: LTLSPEC G (((buffer1.NuPread1 = TRUE & buffer1.NuVfaultyReplicaID1 = 0ud3_1)) & r1.Nuplace = NuSSTART ->X (X (r1.Nuplace = NuSRECEIVED_DETECTION_MESSAGE |r1.Nuplace = NuSSTART |r1.Nuplace = NuSINT_REC_BEGIN)))

Again, the other properties are simpler and they are in a format similar to the detector's ones.

K. Detector and Rebooter

Here, we considered a model that would include both the detector and the rebooter. The goal of this design was to check whether it is feasible to verify the behavior of these two components together.

Assumptions:

1) Communications: synchronized buffered channel from detectors to rebooters. Perfect synchronous channel from the masker to the detectors.

2) Handled data: the same as model 2 of the detector and model 2 of the rebooter.

3) Number of components: one dummy masker, two detectors, two rebooters.

4) Operations: the same as model 2 of the detector and model 2 of the rebooter.

Design: roughly speaking, we merged the model 2 of the detector and the model 2 of the rebooter. The point of connection of these two models is after the checking of crashes. This is a simplification of the protocol, which should notify the rebooter as soon as the crash is detected. This choice is justified by the fact that this is the first merged model and therefore it would need further refinement in order to be closer to the reality. The model 2 of the detector assumed that crashes would be handled correctly. Therefore, the checks are assumed to be done. If any fault is detected, the new variable faultyReplicaID will assume the value of the faulty replica. Based on this value, an internal/external detection message is sent to the local rebooter. In particular, if the faulty replica has the same ID as the detector, then an internal detection message is sent. Otherwise, an external detection message is sent. These messages are handled by the buffer and, subsequently, by the rebooter, exactly as already described for model 2.

Verification: clearly, the model's size grows pretty much and therefore BDD verification will issue the state explosion problem. Again, K-liveness allowed to check properties. The verified properties are exactly the same as the two separated models of the detector and rebooter. See the model file for more details.

L. Unexpected Difficulties

A set of unexpected difficulties has shown up during the various phases of the work. Several of them appeared when the model began growing in size.

1) No code reusability and predisposition to future changes: as long as the models were small, this did not pose a real limit. However, as soon as models had to be extended, it was pretty hard to try variants of a single model: any little change of the model required non-trivial effort, due to the high coupling among the components This difficulty was probably partially due to my programming background, that includes, above all, imperative languages.

2) Conversion phases: although this was an understandable method –since BIP and NuXmv are complementary–, we cannot say that it is that efficient. First, it took a long time to understand how to properly rewrite BIP models in order to have an error-free conversion. For instance, it is not obvious that the BIP model should not have any "if" statement before being converted: that was a specific requirement of the conversion tool. What made this even more sloppy was the necessity to rewrite the BIP model through the old syntax, before converting it to NuXmv, and the non-deterministic generation of values in the NuXmv model, after the conversion. Once this process is understood, doing it once is not very time-consuming. However, things get worse whenever a bug in the model occurs: debugging took extremely long time, since, everytime there was an error, any slight modification led to the necessity of converting again to NuXmv and generating non-deterministic values once more. Even if modifications were done directly on the .smv files, most of the times the process would not be any faster, since any little bug in the model would require a huge change in the code. Furthermore, this last approach would lead to have two non-consistent models (BIP and NuXmv), making any kind of future work extremely hard. Conversion phases are the main factors that slowed down the work.

3) State explosion problem: the first model required a lot of time to be implemented: around one month. If we could prevent this, much time would have been saved, since that large model was not so much reusable. However, in the other models, state explosion problem has been mitigated through SAT-based techniques.

IV. Exercised and Acquired Skills

DURING this project, a variegated set of skills was needed in order to proceed during its steps. During the first phase, it was necessary to get an understanding of the Axo protocol. This involved getting an understanding of the provided papers and of the existing implementation. Since Axo is a distributed system, none of these two steps was trivial. During the second one, it was necessary to properly comprehend C++ code –that used several networking primitives– , since it is the language that was used for the implementation. Then, since I was totally new to modeling and model checking, one of my duties was also to familiarize with this discipline. I had to get an understanding of the matter from a theoretical point of view, including the syntax used to express properties (LTL and CTL). I learnt the BIP and NuXmv modeling languages, along with the related tools. During the whole project, debugging skills were the most valuable ones: the code was more likely not to work than work. No advanced editors are available for the used languages, and many times error messages were not really self-explanatory (especially during the code-conversion phases or when the behavior was not as expected).

V. Conclusion

WE have shown that we were able to implement the models of the detector and the rebooter.
Besides my unexperience in the field, many activities that were performed have been successful, thanks to the work that was performed before the beginning of the project (to acquire background knowledge) and thanks to the provided supervision of this project. In fact, two of the most important components of Axo have been modeled and validated, which perfectly fits with the prefixed goals. Although the level of refinement may be improved, the implemented versions still allow to verify properties that relate to the fundamental functioning of the protocol. Unfortunately, besides the already mentioned unexpected difficulties, which could have been hardly avoided, there have been some other factors that had a negative impact on the result of this work. Mainly, my unexperience with model checking slowed down some phases. Also, it made me introduce some errors that could have been prevented and, sometimes, it pushed me to focus on points that at the end gave no significant result. One example of this is the merged model of the detector and the rebooter. Although, thanks to SAT-based algorithms, properties have been verified, the process could have been speeded up if some errors were avoided and now more refined models would have been available. Another example of mistakes that could have been avoided is the super-refined model, which required a lot of time but did not really provide significant results in terms of model checking. However, in this case, we have a good reference for future implementations. Therefore, for modeling purposes, this model was still a good achievement.
In conclusion, although not everything was perfect, we are satisfied by the obtained results, since the goal of the project has been achieved.

A. Future Work

By keeping in mind the events of this project, there is more than one possible development. One very useful thing to do would be the facilitation of the development of large models, in order to reduce the human effort. As previously pointed out, this caused the work to be significantly slowed down. For this purpose, the most effective solution would be to integrate a GUI (Graphic User Interface) for building the finite state machines (possibly generating NuXmv code directly). Subsequently, continuing the work of refinement would be much more easy and straightforward. As an alternative, other tools may be tested. Models can be always improved and therefore refinement is something that should definitely be done in the future. The provided models are a valid point of start for this purpose. If any of them should be reused in the future, make sure to get an understanding of section II-C, in order to be able to keep the size of the models limited. For the rebooter, it could be nice to simply handle every message (both detection and recovery) through a different atom: we would need an handler for the detection messages, one for the internal recoveries and one for the external recoveries. Another idea could be to improve the merged model: it would be more realistic to have the communication between detector and rebooter as soon as a fault occurs.

Appendix A
About Axo

Although that there are specific papers related to Axo functioning (9 and 10), here we synthesize its main characteristics that are useful for understanding the work. Axo has two main components, that characterize its behavior: controller and masker. The first one receives measurements directly from sensors. It has a library that is made of three software components, namely the tagger, detector and rebooter. From the received measurements, the controller generates a setpoint, which is sent to the PA (Process Agent). The PA has a software library that comprehends the masker. Based on the setpoint, the PA issues a feedback message that is called Validity Report. This feedback is sent back to the controller, but to another component of its library: the detector. The detector has a database that relates to the status of replicas (one entry per replica). This database is updated accordingly to the received report: if the report relates to a new setpoint, then the detector needs to update its database. After this step, the detector performs some checks in order to see whether there is some kind of fault. In particular, by examining its database, the detector determines whether the replica that issued the setpoint is delay-faulty or crash-faulty. This is done by examining the timestamps. Whenever a fault occurs, the detector communicate with the local rebooter, which handles this task: it takes care of both internal and external recovery

processes. Again, this is based on checking timestamps in order to determine whether a replica should be rebooted or not.
One very important characteristic of Axo is that the replicas of the controller do not rely on consensus: they directly communicate with the PA. This allows Axo to tolerate more faults than many other active replication protocols. It also means that the fact that a controller has multiple replicas is not masked from the PA.

APPENDIX B
PERFORMING MODEL CHECKING

The goal of this section is to illustrate how model checking was performed in this project. We show the main commands that were used. For a complete explanation refer to the NuXmv manual [7].
nuXmv -int file.smv: open a model in interactive mode. It is the first thing to do, in order to perform model checking
go: initializes the system for verification
build_boolean_model: Compiles the flattened hierarchy into boolean Scalar. It allows symbolic model checking through SAT-based algorithms
check_ctlspec: verify a property expressed through CTL syntax
check_ltlspec: verify a property expressed through LTL syntax
check_ltlspec_klive: verify a property expressed through LTL syntax, according to a K-liveness approach
We expressed the properties through CTL syntax at first. Then, if we had the state explosion problem, we expressed them through LTL syntax and used the SAT-based algorithm to check them. Most of the time, this procedure solved the issue. The properties are included in the .smv models: CTL properties are written as "SPEC XXX" and LTL properties are written as "LTLSPEC YYY". They can be shown in the interactive mode with the command "show_property". In order to verify them all at once, the command "check_ctlspec" can be used to verify all CTL properties in the database and the command "check_ltlspec_klive" can be used to check all LTL properties. They can also be used to verify a single property.

A. NuXmv nomenclature

These are the main constructs that appear in the generated NuXmv models.

- MODULE: it represents an atom of the model.
- VAR: it contains the variables of an atom.
- INIT: actions that are done when the model is built.
- INVAR: what always holds.
- TRANS: a set of current state / next state pairs. This is what makes the state of the model change.
- NuVname: it is a variable called "name" (in the BIP model). It belongs to the set of VARs.
- Nuplace: it is a variable that represents the state of an atom. It belongs to the set of VARs.
- NuSNAME: it is a state called "NAME" (in the BIP model). These are the values assumed by the variable "Nuplace".
- NuPport: it is a port called "port" (in the BIP model) and it represents a transition of the finite state machine. It belongs to the set of VARs.

For properties verification (just few examples, for more information check the links below):

- AG p: it means **A**lways **G**lobally. Therefore, the property p will have to hold always in all paths.
- G p: it means that the property p holds always.
- AX p: it means that in all paths, the property p holds in the ne**X**t state (of the model!).
- AF p: it means that in all paths, the property p eventually holds.
- EF p: it means that there **E**xists at least on path in which the property p eventually holds.
- ->: logical implication.

Properties are expresses with both CTL [8] and LTL [9] syntax.
For the names of the components:

- detector: drX or dX, where X indicates the ID of the replica.
- rebooter: rX, where X indicates the ID of the replica.
- bufferX: where X indicates the ID of the replica owning the buffer.
- hX: handler of external recoveries on replica with ID X. These recovery messages come from replicas with ID different from X.
- masker: mX. It is just a dummy component.

APPENDIX C
DETECTOR: SUPER-REFINED MODEL

Assumptions

1) Communications: they are assumed to be perfectlty synchronized. This means that whenever a component sends a message to more than one destination –for instance a broadcast– it is assumed that every component will finish to process the message before another one is sent. This assumption is an important characteristic of this model, since the complexity is significantly reduced: the model does not have to handle streams of data through buffers.

2) Handled data: everything is handled as we would expect from reality. Timestamps, health values, identifiers, among others, are the data handled by the database and they are represented as integers. Each one of these integers is represented by a specific number of bits, which is not necessary low.

Design: every operation that related to the behavior of the detector was considered. Therefore, it was necessary to represent the main steps that occur whenever a validity report is delivered: the update of the detector's database, the checks for faults of any kind –crash faults, controllers' faults and detectors' faults– and the communication with the local rebooters.

The components that were put in this abstraction are a dummy masker and the detectors. We recall that the dummy masker is needed in order to trigger the events on the detector –which would act when it receives something from the masker.

[7] https://es-static.fbk.eu/tools/nuxmv/downloads/nuxmv-user-manual.pdf

[8] https://en.wikipedia.org/wiki/Computation_tree_logic

[9] https://en.wikipedia.org/wiki/Linear_temporal_logic

The detector was represented by two BIP atoms: one representing the detector itself and one representing its database. This choice was made in order to reduce the coupling among these two elements: all what regarded the database would be isolated as much as possible, since, as previously stated, we prefer scaling over connectors.

The **database** contains the information that is managed by the detector and that is updated thanks to the validity reports. Here, all the fields that determine the functioning of the detector were represented and contributed to the behavior of the model. Since it was necessary to implement a model that was based on a fixed number of components, the choice was to model three detectors: this would ensure a limited complexity and a sufficient number of replicas to perform meaningful checks. This would imply that the database of each detector would memorize information about the other replicas as well. As already stated, the most important operations of the detector are three: the reception of a validity report, the update of the database and the faults' checking.

The **reception of validity reports** involves the dummy masker and the detector. In this model, the dummy masker would generate non-deterministic reports (since we make assumptions on the masker) and the detector would behave accordingly. All the values of the validity report characterize it and therefore each new report needed to have fresh values. For the sake of simplicity, due to the assumption that we made on the channel, the communication among the masker and the three detectors was straightforward: the masker generates a report, it sends to the detectors via broadcast, and it generates a new report; when all detectors end processing that report, the masker would be able to send a new one to all of them. This assumption made things more simple, since it avoided the implementation of buffers –which made the model more complicated and was considered in other abstractions.

The **update of the database** is performed only when a new validity report is received. Whether a report is new or not is defined inside the report itself, through the controller timestamp value. This phase is quite complicated if a certain level of refinement is desired: it is necessary to handle different kinds data –such as timestamps and health values– which are not very easy to manage in a modeling language, because any kind of operation –or communication with another component– may require one or more ports/connectors. In fact, all what can be simply performed in a conventional programming language, such as "if" statements, in this case may require several lines of code. Furthermore, handling data that can assume several values can lead to make the model even more heavy, since this will increase the number of reachable states dramatically. This is why the model grows exponentially, even if only simple operations are performed.

The **faults' checking** phase consists in performing checks for delay and crash. The first category of faults is based on the health value: if a faulty validity report is received, the health value is decreased. Otherwise, it is increased (if it is not set at the maximum value). Whenever the health is below a fixed threshold –i.e. a certain number of faulty validity reports is received– a delay fault is detected. This triggers a communication with the rebooter, which is considered in another model. Turning to crash faults of replicas, they are detected whenever too much time has passed since a record related to that replica has been received. It is important to notice that in both of these cases, as it can be seen in the BIP code, all operations are detailed and the refinement level is very high. Ideally, this model, with some further improvement, would really allow to verify any kind of property, since everything that relates to the detector is included. However, in practice, this is not feasible.

Verification: the main problem with this model is that adding ports and connectors does not only mean adding lines of code, but also means adding states to the model. This increases the memory complexity as well, and the **state explosion problem** occurs.
In order to try to mitigate the problem, some measurements have been taken. One of them was reducing the number of possible states by simply reducing the range of values: timestamps and health's length were decreased by several bits. However, this was not enough to avoid state explosion: the model could not be even loaded by the model checker.

APPENDIX D
MODEL CHECKING

MODEL Checking is a rigorous way to check whether the model of a system behaves correctly. It consists in abstracting a software or hardware system through a finite state machine, and subsequently check whether this model meets the given specification. The main features of Model Checking, which make this method substantially different from other techniques – such as software testing – , are:

- **Expressional power**: depending on the tools and languages that are used for modeling, it is possible to have a large variety of ways to express the characteristics of a system. For instance, it is extremely useful to define properties in terms of LTL and CTL statements, because it makes possible the verification of many different properties, both **safety** and **liveness**.
- **Flexibility**: through Model Checking it is possible to consider parts of an algorithm, rather than the whole system. This feature allows, at any given time, to focus on important aspects. In fact, this means making **assumptions** on what is less significant or simply too hard to check. Therefore, a very common practice is to isolate parts of the model, assume that some of them behave correctly, and then perfom validation and verification on some other component. This is repeated more than once: there may be a phase that assumes that an element of the model is correct in order to check some statements. A successive phase could consist in verifying the assumptions that have been made previously. After that, results can be put together. This practice is quite common when the system is complex and the resulting model is large.
- **Completeness**: many techniques relies on building test suites, which are made of multiple test cases, in order to **try** to prove correctness. These procedures only provide an approximation of correctness, due to the fact that

they consider a finite, even if large, number of inputs. Model Checking, given a model, allows to build the set of all possible states and subsequently to perform checks on them. In this practice, all the combinations for the inputs are considered. This means that if a property is satisfied, it can be considered a truthful indicator on the correctness of the model. This is useful when it comes to the verification of properties: due to the way in which they are defined, their verification require some formal proof and simply testing some input with some other technique is not enough.

As the name suggests, there are two phases involved in this methodology, whose name is self-explanatory: modeling and checking. The former consists in abstracting a system, and the latter means verifying properties through specific tools called **model checkers**. As already mentioned, the chosen tools have a huge impact on the characteristics of the model and the possibility to check specific properties. In fact, depending on what is used, it might be possible to straightly verify interesting properties. However, it may also be the case that a good language for modeling is not good for checking statements. As section II shows, in this project we had a middle point: the chosen language was suitable for modeling and it also provided the possibility to perform model checking through external tools. However, it was limited to safety properties and therefore we opted for an alternative solution, in order to achieve more expressional power and being able to verify liveness properties too.

APPENDIX E
ABOUT MODELING DETECTOR

Modeling this component was the first phase and it took much time to understand how to properly model this. As pointed out in II-C, refinement is generally preferred over abstraction. Therefore, we believed that it would have been worth spending time in implementing a BIP model of such kind. This why the first approach was to consider an existing Axo implementation, and create a model that had the same level of detail in terms of performed operations: this model would make few assumptions and it would include the features that anyone could expect from an usual implementation C. Since we are considering validation for the detector, several assumptions have been made on the rest. Basically, the detector receives a validity report from the masker, containing some data, and subsequently performs some operations based on that data. Several BIP models have been implemented in this project. However, among them, five were the most significant. One common characteristic of the realized models is that the phase of setpoint generation is skipped. It means that, when we analyze the behavior of the detector, we only focus on what the detectors does, whenever it receives a validity report. Therefore, there is a component –that we called ”dummy masker”– that generates valid inputs for the detector. Since the model has been realized before modeling the rebooter, besides the assumptions that were done on the rest of the protocol, we also assumed that the rebooter would behave correctly. Therefore, as long as the detector could correctly communicate with the rebooter, the model would have been correct.

One very important point, that can get tricky under some circumstances, relates to the correct non-deterministic generation of the values. Since some assumptions are made on the some components of the protocol, we need a way to have all the possible values for the variables that are generated by these components. This is obtained through the elimination of the pieces of code that contain assignments to these variables. In particular, the following variables had to be non-deterministically generated, in order to properly verify the properties:

- The replica ID of the validity report
- The controller timestamp of the validity report
- The detector timestamp of the validity report
- The validity flag of the validity report
- The maximum controller timestamp of the database, whenever a validity report is received. This is necessary, since the timestamps are represented in a simplistic way and therefore the maximum value is just a flag that says whether the report is signaling a crash fault or not. Attemps to make this part closer to reality have been done, in the model with maximum refinement. However, they have been among the major causes that contributed in increasing the complexity.

Another key role is played by the invariants. Besides ensuring the correct behavior of the model, they also guarantee that only meaningful cases are examined and that only the necessary states are computed, if they are used smartly. This mitigates the state explosion problem and avoids useless checks. One example may help in understanding the meaning of this: if we consider the replicaID variable of a validity report, we know that if our model considers three detectors, only three values can be assumed by this variable. Therefore, it makes no sense to consider all the possible values for this variable, which depends on the number of bits that the tool considers for the variable. Thus, restricting the domain can be safely done, in this case. The details about how the values are non-deterministically chosen and how invariants are specified can be found in the implementation of the models (since it is not done in the same way on all models).

A. Properties

This subsection explains with a little more detail how to interpret the properties that are verified in the models. All the properties are contained in the NuXmv models, on the github repository [10]. This explanation examines those files.

Model 1

- There is a first block of statements that simply relates to the synchronization of states between the detector and its database.
- The second block of properties shows that the report is eventually processed.

[10]https://git.epfl.ch/polyrepo/private/files.go?repositoryId=11051&branch=BIPMarco

- The third block makes sure that each state is succeeded by the right one and therefore the report processing is performed well.

Model 2

- The first block verifies that if the database of the detector is updated, then it means that the validity report that was received was a new one. Note that NuVone and NuVzero simply mean 1 and 0, respectively
- The second block verifies that if the received validity report is new, then, if the state of the detector changes, the database will be updated. We cannot omit the option that the state of the atom remains the same, because the model is made of multiple atoms.
- The third block checks that each state is succeeded by the right one.
- The fourth block shows that the report is eventually processed.

Model 3

- The first block contains a set of statements that were useful for debugging purposes
- The second block relates to the detection of delay faults: we check that if the number of consecutive faulty reports is the maximum (i.e. the health is below the defined threshold) then the replica is detected as delay faulty. We also check that the counter of faulty reports is reset after that.
- The third block relates to the detection of crash faults.
- The remaining blocks are similar to the ones of the previous two models.

Model 4

Only the buffer was introduced. The properties are the same as model 3, but they are expressed through an LTL syntax.

APPENDIX F
ABOUT MODELING REBOOTER

We considered a simplified version of the algorithm proposed in the paper 10, rather than trying to reproduce the same level of detail. Since we focus on the rebooter, we make strong assumptions on the detector and we treat it as a dummy component that sends detection messages. One common characteristic of all the realized models is that no messages are lost when two components communicate: therefore we assume that whenever the rebooter sends an external recovery message, it will always receive an ACK. We consider two (dummy) detectors and two rebooters. The values that had to be non-deterministically chosen in the NuXMv model were the ID of the faulty replica and the timestamp at which the fault was detected. Clearly, the non-determinism generation of this value occurs when the detection message is generated on the detector.

A. Properties

The following explanations relate to the NuXmv models that are loaded on the github repository [11].

Model 1

- The first block was useful for debugging purposes
- The second block verifies properties about internal and external recovery processes
- The third block is the same as the second one, but it relates to the other detector
- The remaining statements ensure that the detection messages sent from the detectors are coherent with the recovery processes that are started on the rebooters

Model 2

- The first block makes sure that whenever an internal or external recovery process begins, the ID of the faulty replica, received by the detector, is coherent with the kind of fault that was detected (internal or external).
- The second block verifies some properties about the buffer, which takes the place of the detector.
- The third block relates to the recovery processes, but the handler of the external recovery messages is also included

Model 3

- The first block verifies some properties about the buffer, which takes the place of the detector.
- The second block contains simple properties, which include the handler for external recoveries and the buffer
- In the third block there are properties that relate to the handler of external recoveries. It is also verified that the replica is rebooted when it has to.
- The fourth block verifies that the states are meaningfully traversed by the rebooter atoms. It is also checked that the replica is rebooted when it has to.

[11] https://git.epfl.ch/polyrepo/private/files.go?repositoryId=11051&branch=BIPMarco

REFERENCES

[1] Kenneth L. McMillan, *Symbolic Model Checking: an approach to the state explosion problem*, 1992.
[2] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu, *Symbolic Model Checking without BDDs*, 1999.
[3] Moshe Y. Vardi, *Branching vs. Linear Time: Final Showdown*, 2002.
[4] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu, *Bounded Model Checking*, 2003.
[5] Aaron R. Bradley, *Understanding IC3*, 2012.
[6] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta, *Verifying LTL properties of hybrid systems with K-LIVENESS*, 2014.
[7] Kenneth L. McMillan, *Interpolation and SAT-based Model Checking*, 2003.
[8] Nina Amla, Xiaoqun Du, Andreas Kuehlmann, Robert P. Kurshan and Kenneth L. McMillan, *An Analysis of SAT-based Model Checking Techniques in an Industrial Environment*, 2005.
[9] Maaz Mohiuddin, Wajeb Saab, Simon Bliudze, Jean-Yves Le Boudec, *Axo: Tolerating Delay Faults in Real-Time Systems*, 2016.
[10] Wajeb Saab, *Axo: Detection and Recovery for Crash and Delay Faults in Real-Time Control Systems*, N/A.
[11] Sergio Yovine, *BIP: Language and Tools for Component-based Construction*, 2015.