The NEO Protocol for Large-Scale Distributed Database Systems: Modelling and Initial Verification[⋆]

Christine Choppy[1], Anna Dedova[1], Sami Evangelista[1], Silien Hong[2], Kais Klai[1], and Laure Petrucci[1]

[1] LIPN — Laboratoire d'Informatique de l'Université Paris Nord
99, av. J-B Clément, 93430 Villetaneuse, France
{firstname.lastname}@lipn.univ-paris13.fr
[2] LIP6 — Laboratoire d'Informatique de Paris 6
104 av. du Pdt Kennedy, 75016 Paris, France
silien.hong@lip6.fr

Abstract. This paper presents the modelling process and first analysis results carried out within the NEOPPOD project. A protocol, NEO, has been designed in order to manage very large distributed databases such as those used for banking and e-government applications, and thus to handle sensitive data. Security of data is therefore a critical issue that must be ensured before the software can be released on the market.

Our project aims at verifying essential properties of the protocol so as to guarantee such properties are satisfied. The model was designed by reverse-engineering from the source code, and then initial verification was performed. This modelling work requires choices of adequate abstraction levels both at the modelling and verification stages. In particular, the overall system is so large that the model should be carefully built in order to make verification possible without getting too far from the actual protocol implementation. This paper focuses on the modelling and initial validation of the election process launched at the system initialisation.

1 Introduction

The evolution of nowadays systems is characterised by an increasing complexity and an increasingly critical role. E-government, banks, internet information systems, commerce registries, ... Computers play an essential and growing role in these sensitive sectors and must access and maintain huge databases. In these activities, where the slightest defect may lead to a disaster, safety is paramount. They are characterised not only by the huge amount of data they manipulate, but also by a mandatory high level of security and reliability. The development of such applications is a complex problem which requires to elaborate reliable and safe distributed database management software. Thus, it is advisable to use formal description techniques to clearly specify the behaviour of the considered applications. It is also recommended to have automatic or semi-automatic verification tools to validate these applications.

ZODB, the *Zope Object Database* [4], has become within a few years the most used object database. This free software, associated with the Zope application server is used

⋆ This work is supported by FEDER Île-de-France/ System@tic—libre software.

for a Central Bank, to manage the monetary system of 80 million people in 8 countries [7]. It is also used for accounting, ERP, CRM, ECM and knowledge management. It is now a major free software as PHP or MySQL is.

However, the current Zope architecture does not yet handle data as huge as those mentioned above. In order to attain such performances, the architecture had to be revisited. It led to the design of an original peer-to-peer transaction protocol named *NEO*. This protocol must also ensure both safety and reliability, which is not easy to achieve for distributed systems using traditional testing techniques.

The aim of our work is to formally design and check the safety and the reliability of the NEO protocol. Designing an appropriate specification is a first challenge. Starting from the protocol description, a reverse-engineering process allows for extracting step-by-step a corresponding symmetric Petri net model [6]. Since the original program description is very large and well structured, it is mapped to a modular and hierarchical specification. However, in order to mimic different configurations of the cluster architecture deployed, as well as the different roles of the servers involved, w.r.t. the protocol operation, the model must also be highly parametrised. Regarding the verification step, two main problems arise: which properties of the protocol should be checked and are existing model checking approaches efficient enough?

In this project, we are interested in some critical properties of (a part of) the protocol, such as data consistency, fault recovery, detection of bottlenecks. The verification is then ensured using dedicated techniques (with CPN-AMI platform [2]) taking advantage of the characteristics of distributed systems. We experimented several tools exploiting a variety of reduction paradigms like modularity, symbolic representation (i.e. decision diagrams) and parametrisation in order to tackle the well-known state explosion problem. In fact, the NEO protocol is expected to handle clusters of 100 to 10,000 nodes, most of them being dedicated to storage. Therefore, safety and reliability are critical issues and explicit state space techniques have no chance to overcome the explosion of the state space.

This paper is organised as follows. In Section 2 we describe the general functioning of the NEO protocol, and in Section 3 our modelling approach from the code. Our model of the key feature of the NEO protocol, the election protocol, is presented in Section 4 in some detail, including the modelling of a crash, and of exception handling. A preliminary analysis of the desired properties is explained in Section 5, before a conclusion and some perspectives of this work in Section 6.

2 The NEO Protocol

The general context of the NEO protocol was described in [5], and recalled in Section 1. This section informally describes the general functioning of the protocol.

Different kinds of nodes play dedicated roles in the protocol, as depicted by the architecture on Figure 1:

- **storage nodes** handle the database itself. Since the database is distributed, each storage node cares for a part of the database, according to a *partition table*. To avoid data loss in case of a node failure, data is duplicated, and is thus handled by at least two storage nodes.

- **master nodes** handle the transactions requested by the client nodes and forward them to the appropriate storage nodes. A distinguished master node, called *primary master* handles the operations while the *secondary masters* (i.e. the other master nodes) are ready to become primary master in case of a failure of this node. They also inform the other nodes of the identity of the primary master (light grey arrows in Figure 1).
- the **administration node** is used for manual setup when necessary (dashed arrow in Figure 1).
- **client nodes** correspond to the machines running applications concerned with the database objects. They thus request either read or write operations. They first ask the primary master which storage nodes are concerned with their data, and can then contact them directly.

This repartition of roles raises several issues. The physical architecture is highly reconfigurable: nodes can fail and become unavailable, they can restart or new nodes can be added. In all cases a new configuration is computed.

First, the primary master is *elected* among all master nodes. The election part of the protocol is the main focus of this paper. Even though election algorithms are well-known, a complex election mechanism has been designed as part of the NEO protocol so as to handle node failures or arrival of new nodes during the election process.

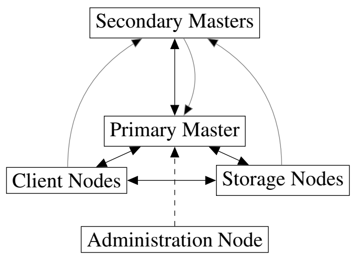


Fig. 1. The NEO-protocol topology

The primary master node maintains the key information for the protocol to operate:

- the **partition table** indicates which parts of the database are assigned to the different *storage nodes*. This allows for duplication which is vital in case of a crash. It can be *updated dynamically* when new nodes join the system or nodes crash. The total number of partitions is fixed when the system starts. The partition table, although maintained by the primary master, is duplicated on all nodes for recovery purposes after a system crash.
- the **last transaction identifier** is used after a system failure to recover a consistent database configuration.

After the election of a primary master, a *verification phase* takes place, checking that all transactions were completely processed, and thus that the database is consistent across the different storage nodes.

Finally, the system enters its *operational state*, where requests from the clients are processed.

3 The Modelling Approach

When the project started, only a prototype version of the protocol was available and no associated RFC (Request For Comments) existed. Therefore, the model we developed relies on the sole implementation.

3.1 Reverse-Engineering

Since the source code of the prototype implementation was the starting point, the modelling task followed a *reverse-engineering* approach.

The source code is now available as free software, under a GPL license. It consists in 24, 200 lines of Python code. The source code follows good programming practices: the code is structured in different files, each of them having a particular concern, e.g. a type of node. The code inside each file contains a few comments giving a description of what the methods are supposed to achieve, with highlights on potential subtleties. Finally, the function, object and variable names are explicit enough to ease the comprehension of the program.

The reverse-engineering approach to the Petri net model construction sticks to the program structure. This allows for:

- focusing on specific parts of the protocol;
- building a structured model, which is a key issue for verification purposes;
- an easy mapping between protocol code and Petri net model, which is necessary when verification results in undesired behaviour;
- a clear separation of concerns between the stakeholders.

Section 4 describes the model of the *election* part of the NEO protocol, including pieces of code and the associated Petri nets (see e.g. Figure 6).

Note that, during the model construction, the code evolved from a prototype implementation to a more robust, distributable version. Some rather important changes hampered our modelling process since a few design choices were revisited. To avoid such problems, we propose in Section 6, *code tagging* to track changes.

3.2 Abstraction Levels

Choosing adequate levels of abstraction is a key issue. First, it is necessary to *select which data structures should be modelled*, and which should not. In the NEO protocol, the nodes exchange a lot of messages. Although it is obvious that the actual data to be stored in the database should not be modelled, it is not always the case for other kinds of information contained in messages, such as the partition tables. These can be exchanged for update purposes, and such messages are important. Modelling these kinds of message easily results in state space explosion.

Second, *the analysis of the protocol is conducted in several stages*: we start by checking that the system without faults behaves as expected ; then faults are introduced to check the recovery procedures after a node failure. Of course, there again, it is necessary to *identify the relevant phases of the protocol for the fault to occur*. If a failure is considered to be susceptible to happen anytime, we face state space explosion without gaining additional relevant behaviour.

The choices of abstraction levels are illustrated for the *election of primary master* part of the protocol, in Section 4. Moreover, Section 4.4 details the addition of faults handling to the model.

4 The Election Protocol Model

The election protocol is a key feature of the NEO system. It is triggered at the bootstrap of the cluster in order to designate among all master nodes a *primary master*. This one will play a central role in subsequent steps of the NEO protocol since it will process requests of clients to access data kept on the storage nodes. After the election, all other masters become *secondary masters*. Their role is only to be ready to replace the primary master in case it crashes. A primary master failure is immediately followed by a new election launched by the secondary master that first detects this failure.

Due to the critical aspect of this component, we developed a detailed model of the election phase in order to be able to simulate it and perform state space analysis. Since the protocol is designed to be (to some extent) fault tolerant we first focused on the ideal scenario where no fault (e.g. a master node failure, a connection loss) can occur.

Although the code of the election protocol is relatively small (around 400 lines of Python code) it turned out to be a tedious task to extract a Petri net model from it. Many high-level data structures are maintained by master nodes for synchronisation issues and a manual slicing step could only remove a small fraction of the code. Therefore, we had to make various abstractions with the primary motivation of obtaining a model amenable to state space analysis.

The Coloane environment [1] was used for modelling. It was chosen for its graphical interface, its analysis facilities using various platforms, e.g., CPN-AMI [2], and Prod [14], and its independence with respect to the underlying formalism for reasons stated in Section 4.5.

4.1 Overview of the Election Protocol and Its Implementation

The goal of the election is to select among all alive masters the one with the greatest *uuid*, a unique identifier chosen randomly by each node at its startup.

The election proceeds in two steps: a **negotiation** step performed by a master node to discover if it is the primary master or not; followed by an **announcement** step during which all masters discover the identity of the primary master and check for its liveness.

Initially, a master node only knows the network addresses (IP address + port number) of its peers provided to it through a configuration file. During the negotiation step it will learn the uuids of all its peers. First it asks the other nodes if they know a primary master by broadcasting an AskPrim message. Other masters answer with an AnswerPrim message possibly containing the uuid and the network address of the elected primary master. The purpose of this first exchange is mainly related to fault tolerance as will be highlighted by the example below. Upon reception of the AnswerPrim message, the master asks its peer its uuid by sending a RequestId message to it. The answer to this message is an AcceptId containing the uuid of the contacted node. This process ends when the master has negotiated with all other master nodes, i.e., it knows the uuid of all its peers. A master node which did not receive any AcceptId message with an uuid greater than its own knows it is itself the primary master.

Note that, although a master may know the identity of the primary before the end of this step (i.e., if it received an AnswerPrim containing the uuid of the primary master) it will still contact other masters. Indeed, the purpose of the election protocol is not *only to negotiate on the identity of a primary master but also to exchange data* that are required for subsequent stages of the protocol.

During the announcement step, the primary master announces to its peers that it is actually the primary master by broadcasting an AnnouncePrim message containing its uuid. Secondary masters wait for this message that they interpret as a confirmation of the existence of an alive primary master. All masters can then exit the election protocol. The cluster is operational and ready to process client requests.

A message of type ReelectPrim may also be sent by a master if it detects a problem during the election, e.g., two primary masters have been designated. Upon its reception, a master will cancel its current work, and restart the election process from the beginning. In a faultless scenario this situation should however not occur.

Figure 2 is a message sequence chart describing a typical election scenario. We only depicted the message exchanges from the perspective of master M2 which will be elected as the primary master. Masters M1 and M3 naturally also have to ask for the same information. In the first exchange M2 asks M1 and M3 if they know a primary master (messages AskPrim). Since all masters are executing the protocol, the primary is not yet designated and they answer to M2 with a AnswerPrim(nil) message. M2 then requests from M1 and M3 their uuid (messages RequestId) which are sent back in messages AcceptId. Upon reception of these two messages, M2 knows it is itself the primary since all uuids requested are smaller than its uuid. It can then announce its election to M1 and M3 using an AnnouncePrim message. Later on, master M1 crashes. After its reboot, it asks M2 and M3 if they know a primary. They both reply that M2 is the master with uuid 897.

This example highlights the fact that entering the election phase is a local decision made by a master at its startup or if it detects the primary crash (or if it loses its connection to it). Thus some master(s) may be in election mode while others are executing the normal protocol.

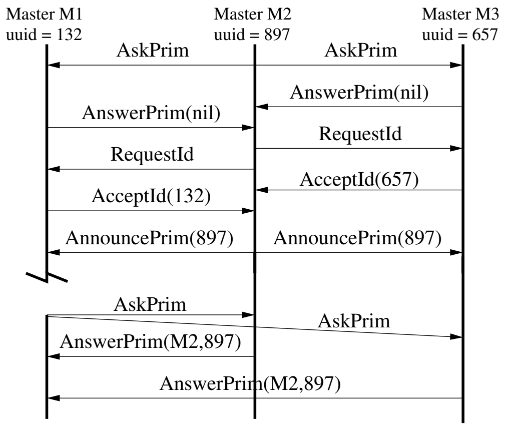


Fig. 2. Message sequence chart describing an election scenario followed by a crash and reboot of master M1

The electPrimary method of Figure 3 implements the election algorithm[1]. It basically consists of four parts: the initialisation of some data structures used in the election process (ll. 2–11); the negotiation part (ll. 12–15); and the code executed by the primary master (ll. 16–21) and by secondary masters (ll. 22–24) as soon as they know their roles.

Among initialised data structures we notice an event manager em (l. 6) used to poll the network, a boolean primary specifying if the node is the primary master (l. 7) and,

[1] As mentioned above, we dealt with 400 lines of Python code which we do not entirely provide here due to space constraints.

```
 1 def electPrimary(self):
 2   self.unconnected = set()
 3   self.negotiating = set()
 4   self.listening_conn.setHandler(election.
         ServerElectionHandler(self))
 5   client_handler = ClientElectionHandler(self)
 6   em = self.em
 7   self.primary = None
 8   self.primary_master = None
 9   for node in node.nm.getMasterList():
10     if node.isRunning():
11       self.unconnected.add(node.getAddress())
12   while len(self.unconnected)+len(self.negotiating)>0:
13     for addr in list(self.unconnected):
14       ClientConnection(em, client_handler, addr=addr,
             connector_handler=self.connector_handler)
15     em.poll()
16   if self.primary is None:
17     self.primary = True
18     for conn in em.getClientList():
19       conn.notify(AnnouncePrimary())
20     while em.getClientList():
21       em.poll()
22   else:
23     while self.primary_master is None:
24       em.poll()
```

Fig. 3. The electPrimary method of the Master class

otherwise, the network address of the primary (primary_master, l. 8). In addition, two sets identify all masters the node is not connected to and has to do so (unconnected, l. 2) or is negotiating with (negotiating, l. 3). The termination of the negotiation step (l. 12) is conditioned by the emptiness of these two sets: at that point, the node has contacted all its peers and received all their uuids. To have a better understanding of the contents of these sets, it seems necessary to mention the different events that have an impact on these two sets (the corresponding code of these events is not fully shown in this paper):

– initially ⇒ m puts in the unconnected set all masters it considers as alive (ll. 9–11 of method electPrimary).
– connection (attempted at ll. 13–14 of method electPrimary) of m is accepted by n ⇒ m moves n from set unconnected to set negotiating.
– m receives an AcceptId message from n ⇒ m discards n from set negotiating.
– m detects the crash of master n ⇒ m automatically deletes n from both sets.

The negotiation is done by repeatedly attempting to connect other master nodes to send them an AskPrim message (ll. 13–14); and handling received messages (l. 15).

Once the negotiation is finished, self.primary is still equal to None if the master did not receive any AcceptId message with a greater uuid and the node knows it is the primary master. It broadcasts an AnnouncePrim message to its peers (ll. 18–19) and keeps treating requests of other masters until all are aware of its existence (ll. 20–21). Otherwise, if the node is a secondary master, it keeps polling the network until it receives the announcement of the primary (ll. 23–24) and then disconnects itself from the primary master (not shown on the figure).

4.2 Model Architecture

The model consists of 18 modules, each of them modelling a specific part of the code. Among them, the most important ones are the three modules listed below.

electPrimary models the method of Figure 3 implementing the election protocol.
poll models the polling method used to wait for and handle incoming packets.
electionFailure models the handling of an exception raised when some synchronisation fault is detected. The election process ends and the masters start a new election.

Modules are dependent and composed using two classical rules [11]: *place fusion* that merges two instances of the same homonym place; and *transition substitution* that refines a meta-transition via its replacement by a subnet modelling the details of the transition. Such subnets always have two specific transitions start and end corresponding to the start and the end of the activity.

In all figures meta-transitions appear in red (see transitions in Figures 5(b), 7(a) — except for transition die, and 8(b)). Guards are put in small notes linked to the the corresponding transition (see Figure 6). Finally, some arc labels, markings or guards are dependent on the parameters of our model although they are automatically generated by a pre-processing of the net. The number of masters was set to 2 in the configuration used for this paper. Lastly, places are coloured in such a way that all instances of the same place have the same colour.

A composition tool has been developed within this project that can assemble several modules through different transformations including the two mentioned above. From several modules and a composition file describing how to combine these, this tool produces a single net resulting from the composition.

An example of composition file can be seen on Figure 4. A composition file consists of a series of net definitions, the resulting net being the last one defined. Other nets are only used to ease the definition of the final one. The figure only depicts the definition of the electPrimary net that can be seen on Figure 7(a). The first step is to define the subnets that will be used to define the net (ll. 4–23). A subnet can be loaded from a file (tag fromFile) or from a previously defined net (tag fromNet). Following these subnet definitions, we have the list of operations performed to produce the net (ll. 24–38). The first operation substitutes the meta-transition poll of net electPrimary by its subnet poll loaded at l. 6 from a previously defined net. The tool is also flexible in that all operations can be conditioned by the definition (or non-definition) of some symbol(s). Here, we can see that transitions crash and primCrash modelling the crash of a master are removed from the net if the symbol faults is not defined (l. 29 and l. 33) when the tool is invoked. The last operation performed (l. 38) fuses all places sharing the same name. It is specified that initial markings of instances are composed using the sum operator. Other possibilities are available: min, max, etc.

Note that the tool is quite generic except for the fuseHomonymPlaces operation which requires to know how the fusion should operate. Hence, the tool is largely independent on the type of coloured nets.

```
 1 <netComposition>
 2   <!-- definition of some nets -->
 3   <defineNet id="electPrimary">
 4     <subNet id="electPrimary">
 5       <fromFile>electPrim.model</fromFile></subNet>
 6     <subNet id="poll">
 7       <fromNet>poll </fromNet></subNet>
 8     <subNet id="secPoll">
 9       <fromNet>secpoll </fromNet></subNet>
10     <subNet id="primPoll">
11       <fromNet>primpoll </fromNet></subNet>
12     <subNet id="sendAnnPs">
13       <fromFile>sendAnnPs.model</fromFile></subNet>
14     <subNet id="sendAskPs">
15       <fromFile>sendAskPs.model</fromFile></subNet>
16     <subNet id="electionFailure" ifdef="faults">
17       <fromNet>electionFailure </fromNet></subNet>
18     <subNet id="crash" ifdef="faults">
19       <fromNet>crash </fromNet></subNet>
20     <subNet id="primCrash" ifdef="faults">
21       <fromNet>crash </fromNet></subNet>
22     <subNet id="reboot" ifdef="faults">
23       <fromNet>reboot </fromNet></subNet>
24     <substituteTrans>
25       <net>electPrimary </net>
26       <trans>poll </trans>
27       <subNet>poll </subNet>
28     </substituteTrans>
29     <deleteTrans ifndef="faults">
30       <net>electPrimary </net>
31       <trans>crash </trans>
32     </deleteTrans>
33     <deleteTrans ifndef="faults">
34       <net>electPrimary </net>
35       <trans>primCrash </trans>
36     </deleteTrans>
37     <!-- substitute other meta-transitions -->
38     <fuseHomonymPlaces method="sum"/>
39   </defineNet>
40 </netComposition>
```

Fig. 4. Part of the composition file for the net of Figure 7(a)

4.3 Detailed Specification of Some Key Elements

General Declarations. Figure 5(a) introduces the main colour classes we have used during the modelling of the election protocol and some places that are shared by all modules of our net. Class M ranging from 0 to MN (the number of master nodes) is used to identify masters with constant 0 specifying a null value[2].

Message types are defined by the MSG_TYPE class. Finally, items of class NEG specify the state of a negotiation between a master m and one of its peers p:

NONE $\Leftrightarrow$ p has not been contacted, i.e., p $\in$ m.unconnected.
CO $\Leftrightarrow$ m has contacted p and is waiting for its uuid, i.e., p $\in$ m.negotiating.
DONE $\Leftrightarrow$ m knows the uuid of p, i.e., p $\notin$ m.negotiating $\cup$ m.unconnected.

Place masterState models the current knowledge that any master m has of the primary master. An invariant property states that for any m $\in$ 1..MN there is a unique token $\langle$m,iam,pm$\rangle$ in this place. Thus:

iam = F $\wedge$ pm = 0 $\Leftrightarrow$ m is a secondary master and does not know the primary.
iam = F $\wedge$ pm $\neq$ 0 $\Leftrightarrow$ m is a secondary master and thinks pm is the primary.
iam = T $\wedge$ pm = m $\Leftrightarrow$ m is the primary master.
iam = T $\wedge$ pm = 0 $\Leftrightarrow$ m is maybe the primary master but is still negotiating.

Tokens in place negotiation specify the content of sets unconnected and negotiating of all masters. For any pair of master (m,n) with m $\neq$ n, there is always a unique token $\langle$m,n,neg$\rangle$ that specifies the current status of the negotiation between m and n as specified above in the description of class NEG.

For each message sent and not treated yet there is a token $\langle$r,s,t,d$\rangle$ in place network with r the receiver, s the sender, t (of type MSG_TYPE) the type of the message, and d the uuid encapsulated in the message (meaningful only if t = AnsP).

Lastly, places electionInit (marked with $\Sigma_{m\in\{1..MN\}}\langle m\rangle$), electedPrimary and electedSecondary model different stages of the electPrimary method: start of the negotiation (l. 12), start of the election in "primary mode" (l. 16) or in "secondary mode" (l. 22).

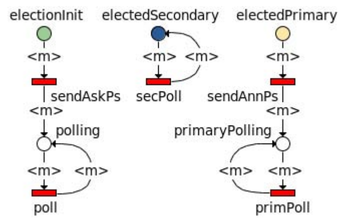
Main Net Modelling the electPrimary method. The net of Figure 5(b) is a high-level view of the electPrimary method. Red transitions are meta-transitions to be later substituted by the appropriate net modelling the details of the transitions. The subnet on the left-hand side of the figure models the negotiation process with the broadcast of AskPrim messages (transition sendAskPs) and the network polling (transition poll). Since we do not consider faults for now, the sendAskPs transition is not in a loop with transition poll: it is useless to retry a connection that can be made at the first attempt. As soon as a master m knows it is a secondary master a token $\langle$m$\rangle$ is present in place electedSecondary. It then keeps polling the network (transition secPoll) until it knows the identity of the primary master. The subnet on the right-hand side models the behaviour of the primary master. Message announcePrim is broadcasted (transition sendAnnPs) and then the primary master keeps processing the messages received (transition primPoll). Note that we do not model here the exit of method electPrimary since messages

[2] Note that we do not distinguish in our model the uuid from the network address. It may however be worth modelling, in a future version, situations where a master reboots and is assigned a greater new uuid, as it may impact on the current election process.

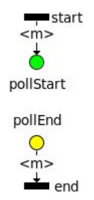
```
parameter
   MN = 2;
class
   BOOL      is [F, T];
   M         is 0 .. MN;
   MSG_TYPE  is [AskP, AnsP, RI, AI, AnnP, RP];
   NEG       is [NONE, CO, DONE];
```

●	masterState	<M,BOOL,M>	<1,T,0>+<2,T,0>
●	negotiation	<M,M,NEG>	<1,2,NONE>+<2,1,NONE>
●	network	<M,M,MSG_TYPE,M>	
●	electionInit	<M>	<1>+<2>
○	electedPrimary	<M>	
●	electedSecondary	<M>	

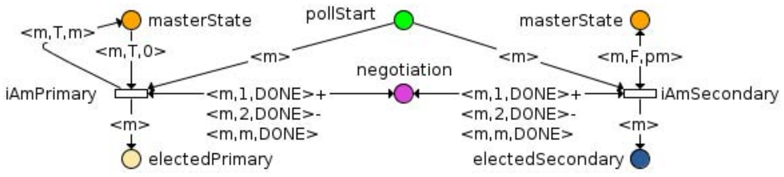
(a) Colour classes and places shared by all modules.



(b) Main net modelling the method of Figure 3.



(c) Model of the poll method.



(d) Master m has negotiated with all other masters and can decide of its role.

Fig. 5. The election protocol model

sent during this election phase may be handled during a call to the poll function made once the server master has exited the method.

A key element of the election algorithm is the poll method called by masters to handle messages received from the network. This method is called with an event manager that is attached several handlers — one for each message type — and only handles a single packet at each call by invoking the appropriate handler. It is modelled by an input transition start putting a token in place pollStart. This place is then merged with homonym places in the handler nets of Figure 6. A token $\langle$m$\rangle$ is present in place pollEnd if master m has finished processing a message. It can then exit the method (transition end). Specifically for the case of meta-transition poll, we also include in its subnet the nodes of Figure 5(d). These model the exit condition of the negotiation step. The negotiation is over for master m if it is not negotiating anymore with any other master: there must not be any token $\langle$m,n,neg$\rangle$ with neg $\neq$ DONE in place negotiation. Depending on the content of place masterState, the token $\langle$m$\rangle$ in pollStart will move to place electedPrimary or electedSecondary — both fused with their homonym places of net electPrimary (Figure 5(b)). If m has not received an AcceptId with an uuid greater than its own (see the corresponding handler in Figure 6), then a token $\langle$m,T,0$\rangle$ is still present in place masterState and changed to $\langle$m,T,m$\rangle$ since m learns it is the primary master (transition iAmPrimary). Otherwise, masterState is marked with token $\langle$m,F,pm$\rangle$ and m knows it is a secondary master (transition iAmSecondary).

Message Handlers. Nets modelling message handlers are presented in Figure 6 along with the corresponding Python code. These nets follow the same general pattern. Their transitions model the reception and handling of messages by removing one token from place network (the message received) and moving one token $\langle$m$\rangle$ (the identity of the receiving master) from place pollStart to place pollEnd, hence specifying the message has been treated and the master can exit the poll function (see the net of Figure 5(c)). The variable s of each transition identifies the sender of the message. Alternatively, the master token can be put in place electionFailed if the processing of the message raises the ElectionFailure exception.

Handlers of types RequestId and AskPrim are rather straightforward. Therefore, we have chosen to focus on types AnswerPrim, AcceptId and AnnouncePrim.

For messages of type AnswerPrim (Figure 6(a)) we distinguish three cases.

– The peer s does not know any primary master (transition handleAnsP1). Local data are not changed by master m that replies to master s with a RequestId message (arc from handleAnsP1 to network).
– Transition handleAnsP2 is fired if s knows a primary (p<>0) and m does not know any or knows the same one (pm=0 or pm=p). The local data of m held in place masterState is updated and, once again, m replies to s with a RequestId message.
– At last, an ElectionFailure exception (ll. 6–9) is raised if m and s both know a different primary master. This is modelled by transition handleAnsP3.

At the reception of an AcceptId message (Figure 6(b)), master m ignores the message if the enclosed uuid s is smaller than its uuid (transition handleAI1) or, if s>m (transition handleAI2), updates its local data by setting its primary field to False (ll. 8–9). In both cases, the content of place negotiation is changed to specify that m has finished negotiating with s: s is removed from the negotiating set of m (ll. 10–11). This will possibly trigger the exit by master m from the negotiation phase and enable one of the two transitions of the net of Figure 5(d).

Finally, a message of type AnnouncePrim can be handled in two ways (Figure 6(c)) depending on the local data of the receiver m:

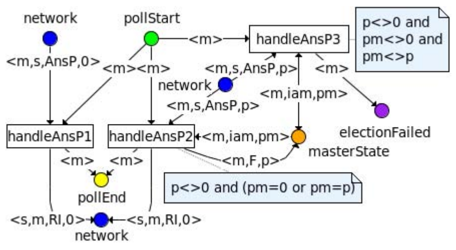
– m does not think it is the primary master. It thus accepts the sender s as the primary master and updates its local data: the token $\langle$m,iam,pm$\rangle$ becomes $\langle$m,F,s$\rangle$.
– m also considers itself as the primary master (ll. 7–8 modelled by transition handleAnnP2) and thus raises exception ElectionFailure.

We mentioned that some synchronisation problems trigger the raise of exception ElectionFailure caught in the body of the electPrimary method. One of the requirements of the protocol is that, in the absence of faults, this exception is not raised. Therefore, in that first modelling step, we left out the handling of this exception and verified through state space analysis that this exception may not be raised.

4.4 Injecting Faults in the Model

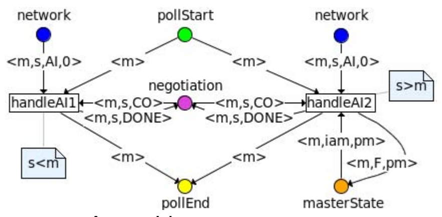
Up to now, we only considered in our models the ideal situation where no malfunction may occur. Since the NEO system is intended to tolerate faults it is a primary concern

```
def answerPrimary(self,
     conn, packet, primary_uuid,
     known_master_list):
  app = self.app
  if primary_uuid is not None:
    if app.primary is not None and \
      app.primary_master.getUUID() != \
      primary_uuid:
      raise ElectionFailure
    app.primary = False
    app.primary_master = \
      app.nm.getByUUID(primary_uuid)
  conn.ask(RequestIdentification(
      NodeTypes.MASTER,
      app.uuid,
      app.server,
      app.name))
```



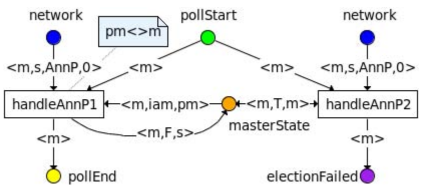
(a) Handler for message type AnswerPrim.

```
def acceptIdentification(self,
    conn, packet, node_type,
    uuid, address, num_partitions,
    num_replicas, your_uuid):
  app = self.app
  # error management
  # ...
  if app.uuid < uuid:
    app.primary = False
  app.negotiating \
     .discard(conn.getAddress())
```



(b) Handler for message type AcceptId.

```
def announcePrimary(
     self, conn, packet):
  uuid = conn.getUUID()
  # error management
  # ...
  app = self.app
  if app.primary:
    raise ElectionFailure
  node = app.nm.getByUUID(uuid)
  app.primary = False
  app.primary_master = node
```



(c) Handler for message type AnnouncePrim.

Fig. 6. Message handlers and their respective models

to enhance our models in order to analyse such scenarios. This injection of faults in the model raises several issues. First, we have to define the nature of the faults we are interested in. Both for modelling and state explosion issues we need to focus on some specific kinds of faults. Second, we must — for the same reasons — abstract the way these faults may occur. If we choose, for instance, to model packet losses, this means focusing on the loss of some specific "strategic" packets, even if any packet may be lost. Last, starting from the faults we choose to model we need to reinvestigate the election program in order to determine which pieces of code that were abstracted away in our first modelling step now need to be considered.

It appeared, during several meetings with the system designers, that the system should be able to recover from the crash of a master. The election protocol should also tolerate other types of faults, e.g., the loss of message, but since most of these are directly handled by lower level layers, they were not considered here. We then decided to restrain the occurrence of such events to two specific situations: the beginning of the election (when any master may be "allowed" to crash), and when a master learns it is the primary master, i.e., when transition iAmPrimary of the net of Figure 5(d) is fired.

The first scenario is the most realistic one: in most cases, the election begins precisely because of a primary master failure. The second one is due to the specific role of the primary master: it announces its existence to other masters, announcement that will cause the exit from the election protocol. Therefore, its failure is a critical event compared to the crash of a secondary master that has (almost) no consequence. As previously mentioned, a look at the election code reveals that these events would typically raise ElectionFailure, exception caught in the main method of the election algorithm. Other exceptional cases are managed in the election code, but most of these deal with errors that are out of the scope of our study, or are defensive programming issues. Therefore these were left out.

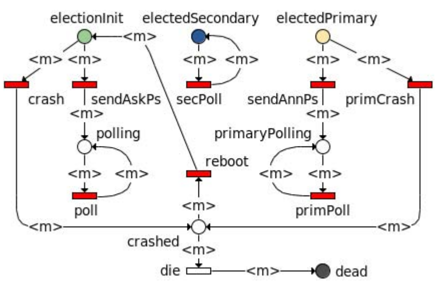
Modelling the Crash of a Master. The net of Figure 7(a) is the main net of Figure 5(b) modified to include the crash of a master. A fault is simply modelled by transition crash (resp. primCrash) moving token ⟨m⟩ from place electionInit (resp. electedPrimary) to place crashed.

After its crash, a master may reboot and join again the election (transition reboot) or be considered as permanently dead (transition die) — at least during the election process. The details of the meta-transition reboot are not given due to lack of space. It consists of reinitialising all the internal data of the master, i.e., the content of places masterState and negotiation, and setting back the token ⟨m,F⟩ in place live (described below) to ⟨m,T⟩.

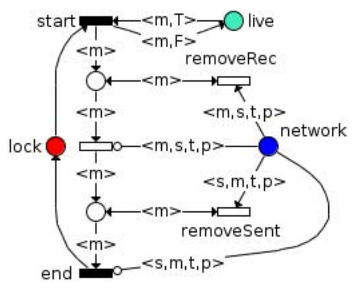
Transitions crash and primCrash are substituted by the net of Figure 7(b) modelling the effect of a crash on the global system. In order to be visible by other masters, a crash must have two side effects. First, the token ⟨m,T⟩ in place live modelling the fact that master m is alive (and considered as such by other masters) is changed to ⟨m,F⟩. Second, the network must be purged from all the messages sent by (or to) master m. Otherwise, if m recovers from its crash, it may handle a message received prior its crash, an impossible scenario that we should not model. Also, a message is automatically ignored by the receiving master if it detects the crash of the sender. So, rather than changing the message handler nets we decided to also purge the network from messages sent by m. This is the purpose of transitions removeRec and removeSent[3]. If transition end becomes enabled, the network does not contain any message with the identity of master m. To guarantee that no message that has to be removed from the network place is received meanwhile by another master we ensured this treatment is atomic by protecting it with place lock. The meta-net of the poll function has naturally been changed in such a way that this lock has to be grabbed before a message is handled.

Faults Detection. The detection by a master m of the crash of one of its peers p is modelled by the net of Figure 7(c). Depending on the state of m this detection has different consequences.

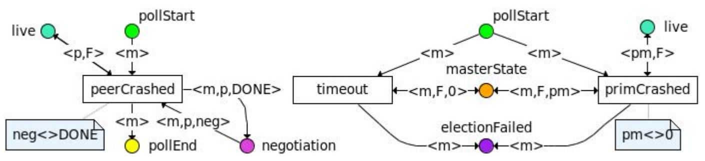
[3] In order to ease the readability we have used inhibitor arcs to check the completion of the network purge. Since the verification tools we use do not support inhibitor arcs, the actual model includes a place counting the number of messages sent by (or to) any master. Zero-test is made via this place. Moreover, note that, due to the additional combinatorics this would generate, we do not model the possibility that a packet is received and handled between a sender crash and this crash detection by the receiver.



(a) The electPrimary method extended with crashes.



(b) Side effects of a crash.



(c) Detection of the crash of a master.

Fig. 7. Insertion of master crashes in the model

If m initiated a negotiation with p and is still waiting for its uuid, it aborts the negotiation as soon as it detects its failure. From the code perspective this consists of removing p from both s.unconnected and s.negotiating. This first situation is modelled by transition peerCrashed that replaces token ⟨m,p,neg⟩ by ⟨m,p,DONE⟩ if master p is dead, i.e., ⟨p,F⟩ ∈ live.

Alternatively, if m is a secondary master waiting for the announcement of the primary master election it can consider this one as dead if it does not receive an AnnouncePrim message after some amount of time. The expiration of this timeout is followed by the raise of exception ElectionFailure. The transition timeout models this second scenario. One of its pre-conditions is the token ⟨m,F,0⟩ to be in place masterState to specify that m is a secondary master not aware of the identity of the primary master.

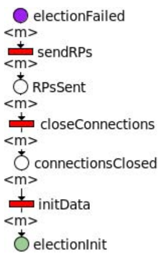
At last, a secondary master m will raise exception ElectionFailure if it detects the failure of the primary master. This is the purpose of transition primCrashed. The master must be aware of the identity of the primary master to raise this exception, i.e., ⟨pm,F⟩ ∈ live (with $pm \neq 0$).

All these transitions are waiting for a token to be in place pollStart to become firable. Hence, they will be included in the appropriate meta-transition of the main net: transition peerCrashed will be put in the subnet of the meta-transition poll while transitions primCrashed and timeout will appear in the subnet of transition secPoll.

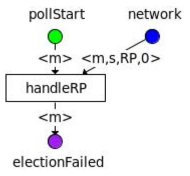
Handling of Exception ElectionFailure. Modelling the handling of this exception is essential if one wants to analyse the election protocol in the presence of faults. Indeed, most synchronisation issues or fault detections will be followed by this exception raise. The code for handling this exception that we had voluntarily hidden in the previous section can be seen on Figure 8(a). It consists of three parts: the broadcast of a ReelectPrim message intended to ask all peers to stop the current election process and start a new one

```
1  def electPrimary(self):
2    ...
3    except ElectionFailure:
4      for conn in em.getClientList():
5        conn.notify(ReelectPrimary())
6        conn.abort()
7      t = time()
8      while em.getClientList()
9         and time() < t + 10:
10       try:
11         em.poll(1)
12       except ElectionFailure:
13         pass
14     for conn in em.getClientList():
15       conn.close()
16     for conn in em.getServerList():
17       conn.close()
18     # restart the negotiation
```

(a) Handler of exception ElectionFailure in the electPrimary method.



(b) Net modelling the exception handler.



(c) Handler of message type ReelectPrim.

Fig. 8. Modelling the handler of exception ElectionFailure

(ll. 4–6); the processing of incoming messages for some amount of time (ll. 7–13); and the closing of all connections (ll. 14–17). After that, the master restarts the negotiation by broadcasting an AskPrim message, waiting for uuids, and so on.

The corresponding net is depicted on Figure 8(b). Its structure reflects roughly the code. The transition sendRps (of which we do not show the details here) puts a token $\langle$n,m,RP,0$\rangle$ in place network for each alive master $n \neq m$. We then close connections (transition closeConnections). The subnet implementing this transition is exactly the one corresponding to the crash of a master (see Figure 7(b)). Indeed, from the viewpoint of another master, closing connections is equivalent to consider the master as crashed. This has the consequence of removing all messages of master m from the network. At last, the firing of transition initData reinitialises the internal data of the master and makes it alive to other masters in order to restart the negotiation. The subnet implementing this transition is the same as the subnet of transition reboot of the net of Figure 7(a). We see that the handler of this exception is quite equivalent to the crash and reboot of a master. We have left out the call to the poll method at l. 11. Indeed, its purpose is mainly to ensure that all peers have received the ReelectPrim message before closing the connections, and to ignore other ReelectPrim messages that could be received meanwhile (see ll. 12–13). Handling other messages is useless insofar as the election will be triggered again. This kind of timing issues need not to be modelled.

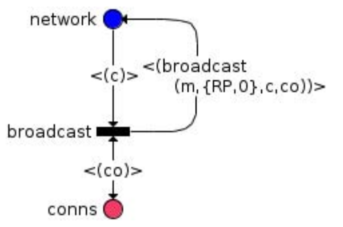
At last, Figure 8(c) depicts the net of the handler of ReelectPrim messages. At the reception of this message a master simply raises the electionFailure exception.

4.5 Alternative Modelling

Although we have presented a symmetric net modelling the election algorithm, we actually created two models of this protocol. The second model has exactly the same module structure but is written in the language of the Helena tool [8]. The purpose of conceiving two models is twofold. First, language Helena is richer than that of symmetric nets. Helena allows, for instance, for the use of list or set types whereas types of symmetric nets are bound to finite discrete types, e.g., enumerate types. The possibility of inserting user-defined functions in arc labels is also a useful way to easily model some problems

```
type id       : range 1 .. 3;
type msgType: enum (AskP, AnsP, RI, AI, AnnP, RP);
type msg      : struct { msgType t; id p; };
type msgList: list[int] of msg with capacity 10;
type conns   : vector [mid,mid] of bool;
type chans   : vector [mid,mid] of msgList;
place network {dom: chans; init: <( [ empty ] )> };
place conns   {dom: conns; init: <( [ true ] )> };
function broadcast (mid s,msg m,chans c,conns co) -> chans {
   for (r in mid)
      if (s != r and co[s,r])
         c[s,r] := c[s,r] & m;
   return c;
}
```

(a) Some type and function declarations



(b) Broadcast of a ReelectPrim message

Fig. 9. Sample of the Helena model

that could, with symmetric nets, only be modelled with the use of additional transitions, irrelevant from a verification perspective. Figure 9 presents a sample of the final model. The place network always contains a single token c of type chans. For each pair of masters (s,r), c[s,r] is the list of messages sent by s to r. The broadcast by master s of a message m is achieved by function broadcast. One of its parameters is the matrix co specifying which masters s is connected to. The broadcast of ReelectPrim messages can then be modelled with a single transition (Figure 9(b)), instead of performing a loop.

This language allowed us to model some features more concisely and to relax some constraints we had with symmetric nets that prevented us from modelling some parts of the protocol. Table 1 lists the features and extensions to the initial model (without faults) with both languages. The connection loss between masters is another type of faults that could be easily modelled in this new model. Although the system is not expected to tolerate such faults, the system designers were still interested to have some feedback on how the system could behave in the presence of disconnections and to which extent it could tolerate such faults. Atomic broadcast is modelled as shown by Figure 9(b). At last, FIFO channels are implemented in the model through a list type.

Table 1. Comparison of the different features provided by both models

	Node crash	Node reboot	Connection loss	Atomic broadcast	FIFO channels
Symmetric Net	✓	✓	×	×	×
Helena Net	✓	✓	✓	✓	✓

We were also motivated by the fact that symbolic and explicit model checking both have their strengths and weaknesses and tools usually implement different reduction techniques to fight the state explosion. For example, although Helena can clearly not manage huge state spaces as a symbolic model checker does, it implements some form of partial order reduction which limits the exploration of redundant executions.

Lastly, this additional modelling effort was relatively small since the modelling tools we use are largely independent of the type of high-level net (this is mainly the case for Coloane, but also for our composition tool used to assemble different modules through place fusion and transition substitution). Therefore, in many cases, we only had to rewrite arc labels from one language to another, an easy task, although a bit tedious.

5 Preliminary Analysis

5.1 Specification of Desired Properties

To ensure the system works as expected, the *desired properties* have to be specified. Hence, the engineers who implemented the protocol provided us with an *informal description* of what they consider as essential properties the protocol should satisfy.

The set of properties we were given contained 70 properties expressed in English. These had first to be refined. In particular, some properties were written in a loose manner that had to be made precise. Some terms had different meanings according to the context. Therefore the statements were re-written in order to make a consistent lot.

Analysis of the different statements allowed for characterising them:

- some of the statements were *not actual properties*. E.g. "the number of out-of-date cells per partition is greater than or equal to 0". Here nothing can be checked since a number of cells cannot be negative.
- the model *does not handle all concerns*. E.g. the data is not modelled and thus it is impossible to prove "data consistency across partitions at all times".
- some statements were *not accurate*. E.g. "there is a positive number of partitions". Actually, this number *is a constant* fixed by the system administrator at the bootstrap and we can easily check that it is never changed.
- part of the properties *concern a particular kind of node*. E.g. "there is no more than 1 primary master node". The master nodes participate in the election of one of them. This election phase must result in selecting a single primary master node. Note that the other types of nodes do not participate in the election, and therefore the verification can be achieved by concentrating on the master nodes model only, as described in Section 4.
- some tricky properties are *expressed on a particular kind of node, but actually also concern the other nodes*. E.g. "there is at least one master node for the system to operate". If all master nodes are down, the system should stop, and thus the other kinds of nodes should not operate anymore. Hence this is a global property.
- some properties *concern a particular stage of the protocol*. For their verification, it is only necessary to focus on this particular part of the protocol operation.

Part of the properties are also only relevant at a specific abstraction level. For instance, we made a distinction between the normal operating mode and the faulty one (see Section 4.4). E.g. "after a primary master failure an election takes place". In the normal operating mode, there is a single election at the system bootstrap. When considering primary master faults, an additional election phase occurs.

5.2 State Space Analysis of the Election Protocol

State space analysis has been conducted on the election model described in Section 4. Symmetric net modules were first assembled to produce a single net describing the protocol. In order to use symbolic tools of the CPN-AMI platform [2], this net was then unfolded in a low-level one using optimised techniques [9] and finally reduced [10] to produce a smaller net (but equivalent with respect to specified properties).

The Helena model briefly described in Section 4.5 was also analysed using a slightly different procedure: since Helena can directly analyse high-level nets, the unfolding step was not performed, and the reduction was directly applied to the high-level net.

For the election protocol we formulated four requirements R0–R3. First, we have seen that, if we do not consider faults, it is important that no exception is ever raised (R0). Two requirements are also logically required for the election protocol (R1 and R2). At last, we want to be sure the cluster can enter its operational state (R3).

R0 - The ElectionFailure exception is not raised.
R1 - A single primary master is elected.
R2 - All masters are aware of the identity of the elected primary master.
R3 - The election eventually terminates.

Both R0, R1 and R2 can be expressed as a safety property while R3 reduces to the absence of cycles in the reachability graph.

Next, we give some elements on the analysis of different configurations we experimented with, and present two suspicious election scenarios encountered.

Analysis of some instances. State space analysis has been performed on some instances of the election model listed in Table 2. It also gives statistics we have gathered on their reachability graphs. A configuration is characterised by the number of masters (column Masters) joining the election, the possibility of observing master crashes (column Crashes), and the number of disconnections that may occur (column Disconnections). The table gives for each configuration the number of markings and arcs of its state space, together with the number of terminal markings, i.e., with no outgoing arcs.

For each listed configuration we have checked whether or not requirements R0–R3 are verified. Our observations are the following ones:

- In the faultless configurations ((2,no,0) and (3,no,0)), the election behaves as expected.
- The possibility of a master crash does not break requirements R1 and R2 but does not guarantee the termination of the protocol even if put aside trivial infinite scenarios during which a master keeps crashing and rebooting.
- Connection loss between two masters is a severe kind of fault in the sense that the protocol does not show any guarantee in their presence. We actually found out very few situations where requirements R1–R3 are still verified despite a disconnection.

Table 2. State space analysis of some configurations

Configuration			Markings	Arcs	Terminal	Analysis Results			
Masters	Crashes	Disconnections			markings	R0	R1	R2	R3
2	no	0	78	116	1	✓	✓	✓	✓
		1	102	165	6		×	×	×
	yes	0	329	650	6		✓	✓	×
		1	434	968	10		×	×	×
3	no	0	49,963	169,395	1	✓	✓	✓	✓
		1	57,526	206,525	52		×	×	×
	yes	0	1,656,002	6,446,764	31		✓	✓	×
		1	2,615,825	10,313,162	84		×	×	×

Faulty Scenarios. The first scenario is quite straightforward and could be discovered by simulating any configuration that includes a disconnection possibility. Let us assume that the protocol is executed by two masters. If they get disconnected, then two elections will take place. Each master is isolated and thus declares itself as the primary master. Some storage nodes will then connect to one master and others will connect to the other master. Hence, there will really be two NEO clusters running separately and the data on the storage nodes will progressively diverge. This scenario is actually not unrealistic if we remember that nodes can be distributed worldwide.

A second suspicious scenario is due to lower level implementation details related to the handling of exception ElectionFailure. It can be reproduced with 3 master nodes M1, M2 and M3. Let us assume that M3 gets elected but crashes immediately after being elected. M2 (or M1) then detects this crash, raises exception ElectionFailure and sends a ReelectPrim message to M1. M1 receives this message and automatically proceeds the same way. Now let us assume that meanwhile M2 closes all its connections and restarts the election before M1 sends its ReelectPrim message. The ReelectPrim message is therefore not received in the handling of exception ElectionFailure (in which case it would be ignored) but after the restart of the election process. This will again cause M2 to raise an ElectionFailure exception, send a ReelectPrim message to M1. If M1 receives its message after it restarts the election (as M2 did), it will proceed exactly the same way. Hence, we can observe situations where M1 and M2 keep exchanging ReelectPrim messages that cancel the current election and restart a new one, thus constituting a *livelock*. The election will never terminate.

Both problems have been reported to the system designers. They are considering some extensions that could prevent the first scenario. It was not clear whether the second scenario is an actual bug or if it is a spurious error due to an over-abstraction in our modelling. Tests were carried out in order to reproduce this situation. Actually, a programming language side effect avoids this problem. The engineers will work on the code to remove this ambiguity.

6 Conclusion and Perspectives

In this paper we presented our work to model the NEO protocol and to verify some critical properties. The modelling is achieved by *reverse-engineering* from the code, and required to devise appropriate choices to work on relevant and useful *levels of abstraction at different steps*. Large applications require a modular modelling with some *composition mechanisms*, and we use the place fusion and transition substitution mechanisms, while a composition tool was developed to put this into practice. We also worked on non-nominal situations by *injecting faults* and on the way *exceptions* are handled, such as the crash of a primary master and checking their adequate detection. While the modelling is achieved using symmetric nets, an alternative modelling in language Helena allows for some additional checks. The work done on the properties started with *analysing the properties provided in natural language* so as to produce relevant and adequate properties that then could be checked.

While this work was going on, the code was changed by the developers (who did not realise that it could impact our work). This raised the *traceability* issue of our work. We plan to address it by "*tagging*" the code with references to the corresponding parts of our model, thus maintaining a *clear correspondence between code and model*.

Further analysis has still to be made on the election process. Developers are gradually integrating various optimisations in the code. Some are really straightforward and should not impact the properties of the protocol while a few are rather tricky and have to be integrated in our model for verification purposes.

Although we only used CPN-AMI and Helena to analyse the election protocol (as detailed in Section 5.2), we plan to use other analysis techniques. This can be done through the facilities provided by Coloane which can interface with several tools: GreatSPN [3] which can build the symbolic reachability graph [6]; Prod [14] which implements the stubborn set technique [13] to avoid the exploration of the full state space; and ModSOG, a modular and symbolic model checker based on observation graphs [12].

References

1. The Coloane tool Homepage, `https://coloane.lip6.fr/`
2. The CPN-AMI Homepage, `http://move.lip6.fr/software/CPNAMI/`
3. The GreatSPN tool Homepage, `http://www.di.unito.it/~greatspn/index.html`
4. The ZODB Homepage, `http://wiki.zope.org/ZODB/FrontPage`
5. Bertrand, O., Calonne, A., Choppy, C., Hong, S., Klai, K., Kordon, F., Okuji, Y., Paviot-Adet, E., Petrucci, L., Smets, J.-P.: Verification of Large-Scale Distributed Database Systems in the NEOPPOD Project. In: PNSE 2009, pp. 315–317 (2009)
6. Chiola, G., Dutheillet, C., Franceschinis, G., Haddad, S.: A Symbolic Reachability Graph for Coloured Petri Nets. Theoretical Computer Science 176(1-2), 39–65 (1997)
7. ERP5. Central Bank Implements Open Source ERP5 in Eight Countries after Proprietary System Failed, `http://www.erp5.com/news-central.bank`
8. Evangelista, S.: High Level Petri Nets Analysis with Helena. In: Ciardo, G., Darondeau, P. (eds.) ICATPN 2005. LNCS, vol. 3536, pp. 455–464. Springer, Heidelberg (2005)
9. Kordon, F., Linard, A., Paviot-Adet, E.: Optimized Colored Nets Unfolding. In: Najm, E., Pradat-Peyre, J.-F., Donzeau-Gouge, V.V. (eds.) FORTE 2006. LNCS, vol. 4229, pp. 339–355. Springer, Heidelberg (2006)
10. Haddad, S., Pradat-Peyre, J.-F.: New Efficient Petri Nets Reductions for Parallel Programs Verification. Parallel Processing Letters 1, 16 (2006)
11. Huber, P., Jensen, K., Shapiro, R.M.: Hierarchies in Coloured Petri Nets. In: Rozenberg, G. (ed.) APN 1990. LNCS, vol. 483, pp. 313–341. Springer, Heidelberg (1991)
12. Klai, K., Petrucci, L.: Modular Construction of the Symbolic Observation Graph. In: ACSD 2008, pp. 88–97. IEEE, Los Alamitos (2008)
13. Valmari, A.: A Stubborn Attack on State Explosion. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 156–165. Springer, Heidelberg (1991)
14. Varpaaniemi, K., Heljanko, K., Lilius, J.: Prod 3.2: An Advanced Tool for Efficient Reachability Analysis. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 472–475. Springer, Heidelberg (1997)