

# From Code to Coloured Petri Nets: Modelling Guidelines

Anna Dedova and Laure Petrucci

LIPN, CNRS UMR 7030, Université Paris XIII  
99, avenue Jean-Baptiste Clément, F-93430 Villetaneuse, France  
[avd.nsu@gmail.com](mailto:avd.nsu@gmail.com), [Laure.Petrucci@lipn.univ-paris13.fr](mailto:Laure.Petrucci@lipn.univ-paris13.fr)

**Abstract.** This paper presents a method for designing a coloured Petri net model of a system starting from its high-level object oriented source code. The entire process is divided into two parts: grounding and code analysis. For each part detailed step-by-step guidelines are given. The approach is illustrated with an industrial application case study, the NEO protocol.

## 1 Introduction

The modelling problem has been under investigation for many years. It features a lot of particular cases depending on 1) the nature of the description of the system to be modelled and 2) which formalism is chosen for the final model. According to the first criteria there are three basic groups of modelling approaches:

1. Starting from an informal description of a problem;
2. Starting from a detailed specification of a system;
3. Starting from the source code.

Some recent works tackle the first group of approaches. For example, in [6] the authors propose a modular design method and illustrate it on a model railway case study. One of the main points of [6] is using properties of the system at the modelling stage. In [8] an approach aggregating different views of the system is given. This method assumes that the system can be observed from several points of view: pre/post, process and lifeline views expressing respectively pre- and post-conditions of events, sequences of events, and sequences of states. Thus, steps in a process view correspond to system events and can be modelled by transitions in a Petri nets formalism. Similarly, steps of a lifeline view correspond to the states of the system and can be modelled by places of a Petri net. Then, by identifying the elements of these different representations of systems, places and transitions are glued together in order to get a complete Petri net.

The second group of modelling approaches includes various attempts to deliver a formal model from UML diagrams [14, 12, 11]. The advantage of these methods is that most developers are familiar with the UML and an automatic transformation of their diagrams into formal models and model-check them, would greatly simplify the software quality control. The difficulty is that UML

diagrams allow for much more freedom for the designer than formal models and the automatic translation is not trivial.

This paper addresses the third group of modelling approaches, which is not covered by a wide range of methods in the literature [16]. Such approaches are dedicated to systems for which the source code already exists, in order to guarantee it satisfies some requirements. They often do not support a complete language, but are restricted to some subset of it. Moreover, to the best of our knowledge, no work addressed a high level object oriented language, such as Python. Some works dedicated to ADA programs focus on the control flow and are limited to boolean variables [9] whereas we consider complex data structures.

Hence, what are the particular difficulties encountered by reverse-engineering from the source code? If a program is rather small (tens of lines) one can simply suppose that the operators are the system events and correspond to transitions, places between them model the intermediate states of the system, and some additional places model the states of variables used. But this approach is no more applicable when the system under consideration is as large as 3 MBytes of object oriented code. Of course it is possible to model all operators as in the previous case, but then the model becomes so huge that there is no means to analyse it and it becomes useless. Thus, it is necessary to choose an appropriate *level of abstraction* for the system. If it is too low and the model contains too many details, the same problem as above arises. If the level of abstraction is too high, there are too many hypotheses and assumptions and it may happen that nothing is left worth checking. The model is then trivial and its behaviour is completely correct while the system contains drawbacks that are hidden due to the modelling assumptions.

This paper reports the process adopted when modelling the NEO protocol using a reverse-engineering approach within the Neoppod project [4, 5]. The project aimed a formally proving that some essential properties of a new distributed database management system were satisfied, thus giving sufficient confidence in its correct behaviour. This allowed us to gain experience on the methodology we used, derive finer and general guidelines for modelling which constitute the core of the paper, and check that the models obtained using these guidelines were as expected. Moreover, this work confirmed our opinion that Petri nets and more specifically high-level nets were an appropriate choice as modelling language for such purposes since they make the model fit quite closely the initial source code. These benefits of using Petri nets for software engineering purposes are exposed in [7]. Thus, even without prior knowledge of Petri nets, the protocol programmers/engineers could easily come to understand the model, which additionally gave them a synthetic overall view of their large program.

The paper is organised as follows. Section 2 gives detailed guidelines on how to derive step-by-step a coloured Petri net from the source code. Then Section 3 shows how this method was applied in practice to the NEO protocol. Finally, Section 4 draws some conclusions.

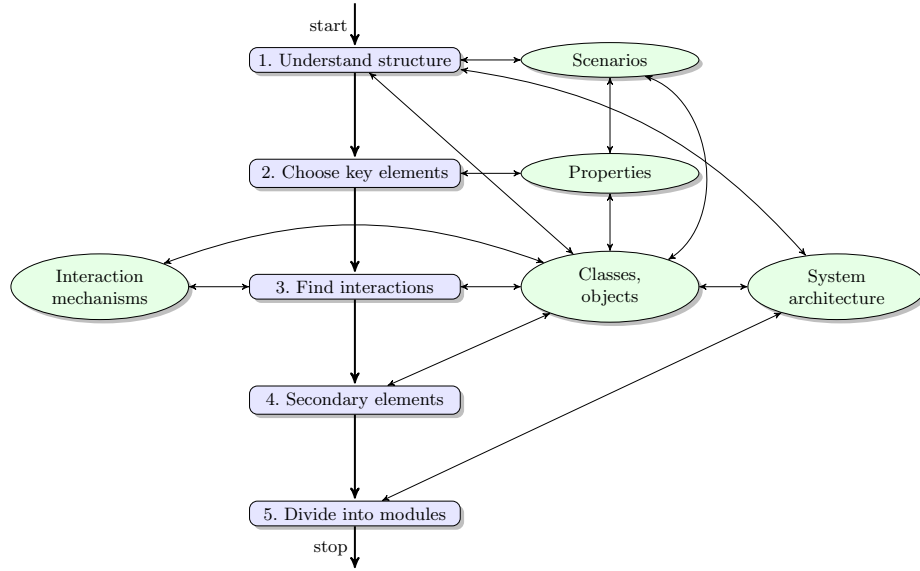
We assume the reader is familiar with coloured Petri nets [13].

## 2 Modelling Guidelines

This section discusses the guidelines to follow in order to deliver a coloured Petri net from high level object oriented source code. These guidelines are illustrated with the NEO protocol in Section 3, and the reader is invited to read both in parallel.

### 2.1 Grounding

Before the start of the modelling process some preparation work is required. It mainly concerns the deep understanding of the project structure and expected properties of the system. This helps a lot during the modelling by saving the time devoted to the consideration of unnecessary elements or restructuring model hierarchy. It is always possible to skip this stage and proceed directly to modelling the most interesting piece of code, but then the risk of choosing an inadequate abstraction level is very high. The main steps of grounding are listed below. They are depicted in Figure 1 together with the elements that are to be considered or are impacted.



**Fig. 1.** Schema of the grounding process

1. **Understand the structure.** First of all, we should pay attention to the architecture of the project. The key elements (classes) should be found as

well as their roles in the whole system. It can be very useful to find the most common scenarios of the system use (or maybe scenarios that should be verified later). We can look for the parts (classes or objects) of the system that are impacted by these scenarios. We also need to understand the class structure of the project (paying a particular attention to inheritance and polymorphism). During this step the most important result is global comprehension of how the system works from the inside.

2. **Choose key elements.** The second step focuses on the system properties to be checked. Properties can be proposed by developers, clients or anyone else. Then, they must be considered one by one in order to choose those that are the most crucial for the system. Selecting them before starting the modelling process is very important since this choice can influence a lot the model structure that will not be so easy to change later on.

Once the properties are selected, we look for the scenarios they concern. Moreover, the classes and methods used within these scenarios are selected, according to the project structure from the previous step. Thus, the main pieces of code that are going to be modelled are defined.

3. **Find interactions.** We should keep in mind that objects of chosen classes can be verified separately from one another. But the ultimate goal is usually to model-check the whole system altogether. Separate parts can be subjected to traditional testing techniques while the complexity and the size of the system makes their application to the entire project impossible.

Hence, when modelling something larger than an isolated object, the interactions between different objects must be identified. These can be of very different natures: message passing; shared or global variables (e.g. between different methods inside a class); sometimes a class is composed of other auxiliary classes; a method of an external class can be called. All interactions should be investigated and the corresponding elements added to our modelling selection.

4. **Secondary elements.** Here we need to look at auxiliary classes that are used by the selected key classes. They can be classes of data structures, or classes providing message exchange capabilities. On the one hand, operations and/or interconnections of key elements are impossible without them. On the other hand, if we model them in detail, the model will be too bulky for analysis to be performed. Such elements usually describe the work of the system on a low level of abstraction and can be verified separately. So, the idea is to model them as simple as possible, but without loss of essence.

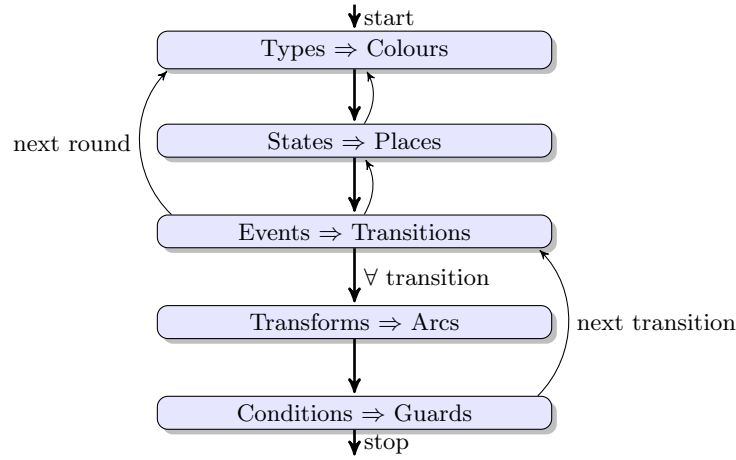
At the end of this step we should know which abstractions are going to be used: some algorithms could be modelled as a single transition, some complex data structures encoded with natural numbers, etc.

5. **Divide into modules.** All the scenarios and methods that have been chosen for modelling are used to design a modular structure for the future model. Of course, it can be changed later during the modelling process.

It is rather natural to associate a submodule with a class or a method. It is also important to pay attention to secondary elements and decide whether they are worth a separate module or not.

## 2.2 Code analysis

During this stage, two processes are carried out in parallel: the analysis of the source code and the construction of the model. In order to streamline these processes we propose to divide them into five main activities, according to the scheme on Figure 2. Each activity requires to look for some elements in the source code as well as interpreting them in terms of the modelling formalism (in our case coloured Petri nets). At each step, the source code is observed from different points of view in order to extract the different components of the model. In practice, it is usually necessary to perform the cycle several times but at the



**Fig. 2.** Schema of the modelling process

start it is hard to tell how many times it should be done. It is also possible that some activities are skipped on later rounds, since a new element cannot be extracted from the source code. From one round to another the understanding of the chosen abstraction level is more and more accurate and the model is more and more complete.

Since the module hierarchy of the CPN can be different from the initial structure of classes and methods, the work within the five activities can be organised in different ways.

- Consider the modules of the future model (found at the fifth step of grounding) one by one. For each module examine scenarios, classes and methods it concerns and analyse them via all activities.
- Consider scenarios or methods (found at the second step of grounding) one by one. For each scenario/method perform each activity that will give refinement for different modules of the model.

- Consider activities one by one and look at the system as a whole, analysing different parts of code and changing different models, but from the chosen point of view.

In practice, the third approach is difficult to apply unless the model is almost ready and can be grasped at a glance. The first approach is the most effective one, but sometimes the second one may also prove useful by focusing on a particular behaviour. In this case, the behaviour is either described by an execution scenario, or the details of a method are tracked step-by-step.

**Data structures** It is important to start from this activity because it forms the basis of the future model. It is natural to start with colour domains in order to use them (and may be enhanced later with new details) during further activities.

In general, data structures of the source code *should* be expressed in terms of colour domains. However, it is often not that simple. In object oriented code, data structures are usually integrated together with their storing, loading and treatment methods. Colour domains syntax does not allow to do this, so, it may be needed to model a “simple” object with a separate CPN. Such cases can be left to further activities nevertheless providing basic types for future CPNs.

This phase provides as a result a preliminary list of colour domains and variables needed in the model.

**States and conditions on objects** This is the first activity that assumes the modeller thinks in terms of parts of CPN that have no strict correspondence in the source code, namely the places. It may be difficult to deliver them in the situation of “blank sheet”, but the model with places make other activities become much simpler or even possible (e.g. construction of arcs).

Hence, this phase aims at creating the set of places of the CPN which usually represent the states of the system or its parts (objects, variables, etc.). To begin with, the system flow of operations can be represented as a finite state automaton. The set of states of this automaton can be a first approximation of the set of places of the CPN. Then conditions required to proceed from one state to another are considered. These conditions often concern the states of some objects or variables. They should also be added to the set of places. Finally, a colour domain (defined during the previous activity) is associated with each place. If some variable or object is directly mentioned in the properties of the system, it can be directly represented as a place on the first round and may be transformed into a group of places or a subnet on later rounds.

**Events and actions** This activity is in general simpler than the previous one. Each operator or method call in the source code can be considered as an action and thus be modelled as a transition. The main hindrance here is a tendency to model every operator with a transition. To avoid this we can apply information obtained during grounding (2nd and 4th steps).

The purpose is to select actions, essential for the processes to be verified. To start with, consider the changes of variables and data structures that are implicitly mentioned in the properties. If the properties are not formalised yet, main constructions of the system can serve as a basis. As for previous phases, on the first round only a preliminary view of transitions in the model can be given. After going through other activities it will be completed and refined.

**Transformation of data** During this phase, the modeller considers for each transition the three following questions, and performs the corresponding net construction:

1. What is taken as input? (Connect corresponding places with input arcs);
2. What is produced as output? (Connect corresponding places with output arcs);
3. How are the tokens transformed? (Provide input and output arc expressions).

If there is a special input format, it can be reflected in the input arc inscription. If the output is somehow calculated from input variables, the corresponding output arc must be assigned with a formula, representing these calculations in terms of CPN. Often, the formulae from the source code cannot be applied directly and need to be adapted w.r.t. the chosen level of abstraction.

**Conditions on events** Here, as in the previous phase, we consider the set of transitions. The focus this time is on the special conditions under which a transition can be fired. In practice the conditions for most transitions are modelled by the matching of tokens in input places with arc expressions. In this case the transition has a guard `true` that can be omitted in the model. But sometimes for better readability of the model, and also to prevent having too large sets of places and transitions, it can prove better to formulate such a condition as a guard of the transition.

The goal of this activity is to find such cases and to figure out the guards. It can happen that some condition is not possible to express on the selected level of data abstraction. If so, the colour domains created in the first activity must be revised, as well as their occurrences in parts of the CPN that have already been built. Thus, an additional round of activities is started.

So, in this section we gave the detailed guidelines to follow in order to model a system starting from high level object oriented source code. In the next section these guidelines will be applied step-by-step to the industrial case.

### 3 Application of the Guidelines to the NEO Protocol

This section illustrates modelling guidelines with examples from modelling process of the NEO protocol. The protocol, designed to handle a large distributed database over a cluster of machines, was described in [4, 5]. Its main characteristics are shortly summarised in Section 3.1. This specification was part of

an industrial project which aimed at validating the protocol and its prototype implementation both designed and developed by the NEXEDI company. It was implemented in Python, but our approach is not specific to this language.

### 3.1 Brief Description of the NEO Protocol

A more extensive description and analysis of the NEO protocol can be found in [4] and [5].

Different kinds of nodes play dedicated roles in the protocol, as depicted by the architecture in Figure 3:

**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. *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 other nodes of the identity of the primary master.

**the administration node** is used for manual setup if needed.

**client nodes** correspond to the machines running applications concerned with the database objects. Thus, they 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.

The life cycle of the NEO protocol is depicted in Figure 4.

At the system start, the primary master is *elected* among all master nodes. The primary master maintains the key information for the protocol to operate.

After the election of a primary master, the system goes through various stages with the purpose of checking that all transactions were completely processed, and thus that the database is consistent across the different storage nodes (*bootstrap* protocol).

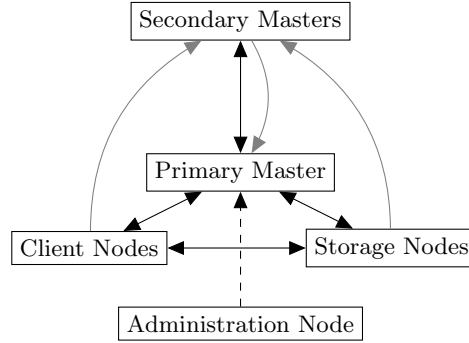
Finally, the system enters its *operational state*. Clients can then access the database through the elected primary master.

As for storage nodes, once they are connected to the primary master, they check the consistency of the information they detain, and initialise their service before being ready to serve requests.

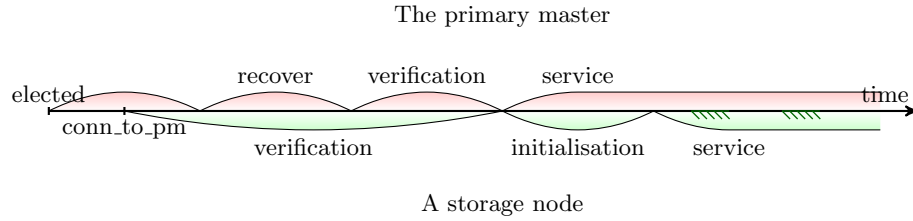
### 3.2 Grounding

Each step described in Section 2.1 is now applied.





**Fig. 3.** Protocol architecture



**Fig. 4.** Phases of bootstrap process

*Understand structure* This step is difficult to illustrate on a real example since it implies working on extensive code. The conclusions cannot be confirmed by a small piece of code. Nevertheless, for the NEO protocol, at this stage we can state the following, and confirm the brief description from Section 3.1.

The main entities are nodes of the cluster, of four types: master, storage, client and admin nodes. For each of these types there is a corresponding class in the source code.

The life cycle of nodes leads them through different phases implemented by an auxiliary class (RecoveryManager, VerificationManager) or a method of the corresponding node class (ElectPrimary, VerifyData, Initialize, DoOperation). Also, depending on the phase of the protocol, a node changes its message handlers.

*Choose key elements* Based on the conclusions of the previous step and on the verification issues, we decided to focus on master and storage nodes. This paper does not get into the details of the numerous properties to check, a large part of which can be found in [4] and [5]. Many properties were provided as an informal statement by the code developers. For example, only a single node is elected as a primary master; all shared information (partition tables, identifiers) has been made consistent for the service phase to take place.

Most attention is paid to the election of the primary master and to the bootstrap process (everything between election and operational state). Later on in this paper we focus on bootstrap phase. For this phase a successful scenario implies that the primary master (supposed to be correctly chosen during election phase) checks that its critical information is up-to-date (recovery phase), verifies the coherence of the unfinished transactions (verification phase) and allows storage nodes to start operation (service phase).

Therefore, we chose the following fragments of code for detailed analysis:

#### 1. master node application

```
1 def __init__(self, config) #initialisation of a master node
2 def run(self) #main life cycle of a master node
3 def playPrimaryRole(self) #describes the behaviour of the primary master
4 def runManager(self, manager_class) #loads a specific manager
5 def changeClusterState(self, state)
6     #changes the state of the whole cluster
```

#### 2. recovery manager class

```
1 def __init__(self, config) #initialisation of the recovery manager
2 def run(self) #describes the main activity of the manager
3 def buildFromScratch(self) #called if the partition table is injured
```

#### 3. verification manager class

```
1 def __init__(self, app) #initialisation of the verification manager
2 def _askStorageNodesAndWait(self, packet, node_list)
3     #called each time when the same message is sent to all SN
4     #and the answers are required in order to continue boot.
5 def run(self) #describes the main activity of the manager
6 def verifyData(self) #verifies consistency of unfinished transactions
7 def verifyTransaction(self, tid)
8     #verifies that different replicas of the transaction
9     #are coherent
```

#### 4. storage node application

```
1 def __init__(self, config) #initialisation of a storage node
2 def run(self) #main life cycle of a storage node
3 def connectToPrimary(self) #connects to the primary master
4 def verifyData(self) #launches verification phase
5 def initialize(self) #launches initialisation phase
```

*Find interactions* The nodes in the cluster need to communicate with one another. For this purpose they use a class EventManager. It describes the mechanism for sending and receiving messages. To treat them, each node has its own handlers, different for the different phases of the protocol. Thus, they should be added to our list of pieces of code.

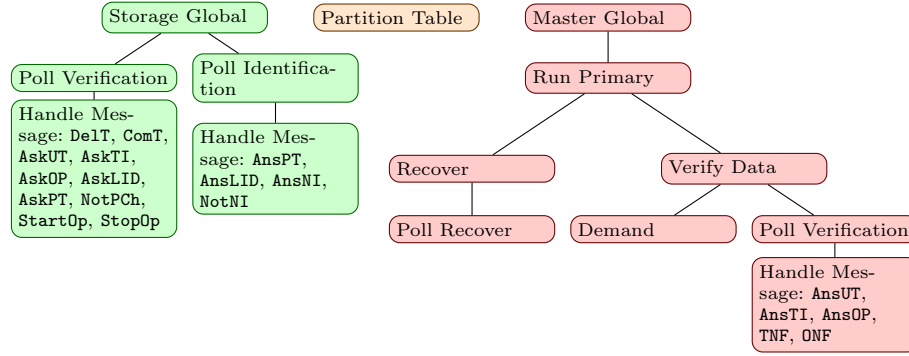
Another means for nodes collaboration in the cluster is a partition table. It is implemented as a class that stores the distribution of data among storage nodes. This class is another key element and should also be added to the analysis list.

The master and storage applications also use some global variables to allow their methods to know the state of the application (primary, operational, has\_pt — partition table). This information should be kept aside to be used during modelling.

*Secondary elements* When this stage occurs, all significant parts of the project and their communications are identified. It is then time to make rather crude abstractions on objects that could not be eliminated from the model, but must be simplified because of the abstraction level.

For example, for the NEO protocol we made following abstractions:

1. The complex message structure, defined in a class package is modelled as an integer number;
2. Connection, described as a group of classes, is modelled as a pair of nodes, that are considered to be connected;
3. Transaction and object of the database, that have a lot of fields, such as serial number, history, data, etc., are modelled by their identification numbers.



**Fig. 5.** Hierarchy of models

*Divide into modules* Figure 5 presents the subnets structure of the bootstrap model. Let us discuss how it was created step-by-step.

The methods chosen for detailed analysis in the second step can be divided into two groups: those concerning a master and a storage node. Thus, nets “Master Global” and “Storage Global” will be constructed starting from source code of methods `__init__` and `run` of the corresponding cluster nodes. The master node net will also include the analysis of `playPrimaryRole` method and the storage node `connectToPrimary` method.

A storage node calls two methods corresponding to verification and initialisation phases. They can become sub-modules of “Storage Global”. During each of these phases, a number of message types is treated. The nets, modelling the handlers of these messages, will be sub-nets of the “Verification” and “Initialisation” nets.

The primary master calls recovery and verification manager during the bootstrap. So, it is natural to model this with sub-modules corresponding to each manager. Thus, the “Recovery” net is the result of the code analysis of methods

```

1 class Cell(object):
2     def __init__(self, node, state = CellStates.UP_TO_DATE):
3         self.node = node
4         self.state = state
5     ...
6
7 class PartitionTable(object):
8     def __init__(self, num_partitions, num_replicas):
9         self._id = None
10        self.np = num_partitions
11        self.nr = num_replicas
12        self.num_filled_rows = 0
13        self.partition_list = [[] for _ in xrange(num_partitions)]
14    ...

```

**Fig. 6.** Fragment of the source code declaration of partition table class

runManager and changeClusterState of the primary master and methods \_\_init\_\_, run and buildFromScratch of the recovery manager class. It has one sub-net that corresponds to method poll used for treating incoming messages. Similarly, “VerifyData” is built on the basis of methods runManager and changeClusterState of the primary master and all methods from verification manager class chosen for detailed analysis. During modelling, this sub-module has become too complicated, so it was decided to replace some parts of it by sub-nets. Thus, method \_askStorageNodesAndWait that is called several times was modelled by a “Demand” net. Treating incoming messages corresponds to “Poll Verification” net with sub-nets, corresponding to handlers of different message types.

As it was found at the third step that the partition table class plays major role in interactions between nodes. Since this class is rather sophisticated, it forms a separate net “Partition Table”.

### 3.3 Code Analysis

In this subsection we will give some detailed examples of application the guidelines from Section 2.2 to the source code of the NEO protocol.

*Data structures* In order to show how the colour domains can be constructed from data structures, let us consider the piece of code in Figure 6. It is a fragment of the partition table class definition where the internal fields are declared. First, the class for a cell of the partition table is declared. It has two attributes: a storage node and a state. Knowing that a partition cell has two possible states (up-to-date or not), we can declare a colour domain PSTATE as a set of these two values and a colour domain PT\_CELL as a product of storage node type and cell state.

```

colset PSTATE = with UTD | OOD; (* the set of states of partition *)
colset SN = index sn with 0..N; (* the set of storage nodes *)
colset PT_CELL = product SN*PSTATE; (* a cell of partition table *)

```

Now let us consider the beginning of the partition table class constructor. It starts with assigning the values of variables for the number of replicas and the number of partitions. Then it creates a two-dimension list. In one dimension its

```

1 def run(self):
2     self.app.changeClusterState(ClusterStates.VERIFYING)
3     self.verifyData()
4     ...
5
6 def verifyData(self):
7     em, nm = self.app.em, self.app.nm
8     neo.lib.logging.debug('waiting_for_the_cluster_to_be_operational')
9     while not self.app.pt.operational(): em.poll(1)
10    neo.lib.logging.info('start_to_verify_data')
11    self._askStorageNodesAndWait(Packets.AskUnfinishedTransactions(),
12    [x for x in self.app.nm.getIdentifiedList() if x.isStorage()])
13    ...
14
15 def _askStorageNodesAndWait(self, packet, node_list):
16     poll = self.app.em.poll
17     operational = self.app.pt.operational
18     uuid_set = self._uuid_set
19     uuid_set.clear()
20     for node in node_list:
21         uuid_set.add(node.getUUID())
22         node.ask(packet)
23     while True:
24         poll(1)
25         if not uuid_set:
26             break

```

**Fig. 7.** Fragment of the source code of the verification phase

size is equal to the number of partitions, in the other dimension the size is not specified. So, we declare three auxiliary colour domains: a set of partitions, a list of partition cells and a partition row, that is a product of a partition and a list of cells. Finally, a partition table colour domain consists of a list of partition rows.

```

colset PART = index np with 0..NP (* the set of partitions *)
colset PT_CELLlist = list PT_CELL with 0..(NR+1); (* a cell list *)
colset PT_ROW = product PART*PT_CELLlist; (* a partition table row *)
colset PT = list PT_ROW with 0..NP; (* the partition table type *)

```

The following colour domain definitions will be used later on:



```

colset MN = index mn with 0..M; (* the set of master nodes *)
colset NODE = union s1:SN + m1:MN; (* the set of all nodes *)
colset CSTATE = with VER | REC | RUN | STP; (* the set of cluster states *)
colset MTYPE = with StopOp | StartOp | AskUT | AskPT | AskNI | AskLID |
                AskTI | AskOP | AnsUT | AnsNI | AnsPT | AnsLID |
                AnsTI | AnsOP | NotNI | NotPCh | DelT | ComT;
                (* the set of message types *)
colset MESS = product MTYPE*NODE*NODE*INT; (* the set of messages *)
colset SNlist list SN with 0..N; (* a list of storage nodes *)
colset MESSlist = list MESS 0..1000; (* a list of messages *)


```

*States and conditions on objects* As an illustration of the next four steps, let us consider the beginning of the verification phase from the primary master point of view. The corresponding source code is listed in Figure 7.



The first three lines come from the `run` method of the verification manager class. We can see that the primary master changes cluster state to `VERIFYING` and calls the `verifyData` method. So, we can start by defining two places:

-  *start\_verif* with colour domain MN (the state of the primary master at the start of the verification manager);
-  *c\_state* with colour domain CSTATE (the current state of the cluster).


Then, the primary master waits until the partition table becomes operational (line 9). We define a new place, corresponding to this state of the primary master:

-  *wait\_pt* with colour domain MN.


After that, it calls a method `_askStorageNodesAndWait`, where it sends requests about unfinished transactions to storage nodes, and waits until the list `uuid_set` becomes empty. This waiting period can be modelled as a new place. In order to send messages to other nodes we need a channel place. According to the protocol, it must be a FIFO list. Hence, two places are added:

-  *network* with colour domain MESSlist;
-  *wait\_ut* with colour domain MN.


Finally, the primary master starts the verification of transactions one by one. This code is out of scope of our example, but we can at least give the next state of the primary master by adding a new place:

-  *verifying\_trans* with colour domain MN.



*Events and actions* Now, we need to extract the important actions from the same piece of code. The first method call `changeClusterState` can be considered as one of them. So, we add the first transition:

-  *change\_c\_state*.


When getting to the next lines, we see that line 7 contains nothing but shortcuts and line 8 writes the current state to the log. The next important action is `em.poll(1)` that is executed while the primary master waits for the partition table to be operational. Here, it is supposed to treat different messages. For the sake of readability of the model, we decide to organise message handlers in separate sub-nets. A new transition is added, coloured in black to symbolise there is a net behind.

-  *poll\_pm\_verif*.

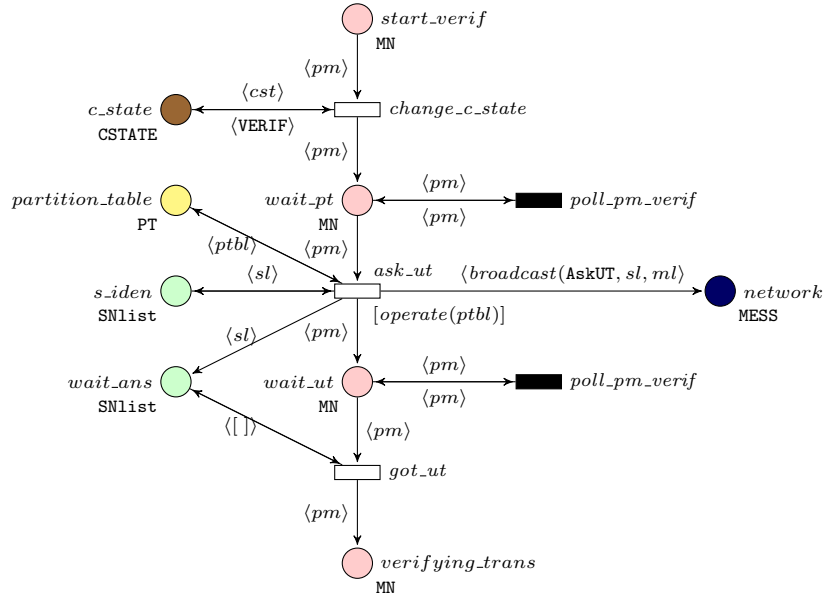
Line 10 is not important since it writes a log. Then the `_askStorageNodesAndWait` method is called with a list of all identified storage nodes as an argument. Inside this method some shortcuts occur (lines 16–18) and the list is cleared `uuid_set`. Then, considering the storage nodes from the input list one by one, they are added to `uuid_set` and send a request for unfinished transactions (which is also given as input parameter, defined in line 11). An additional place is needed to store the identified storage nodes. So, we go to the previous step, add this place, and return to add a new transition:

-   $s\_iden$  with colour domain SN.
-   $ask\_ut$ .

Then the primary master is waiting once again, executing `poll` (line 24). So, we duplicate the corresponding transition. Finally, we add a transition that models the exit of this process.

-   $got\_ut$ .

*Transformations of data* Let us consider the transitions we have up to now one by one in order to build arcs and provide their expressions. To do so, some variables should first be declared. Let  $pm$ : MN;  $cst$ : CSTATE. The whole net can be seen in Figure 8. Transition *change\_c\_state* moves the primary master token




**Fig. 8.** Model of verification manager — part 1

from place *start\_verif* to *wait\_pt* and replaces the current cluster state token with a VERIF one.

Transition *got\_ut* moves the primary master from place *wait\_ut* to place *verifying\_trans*. But it can fire only when all answers are received from the storage nodes. Here an additional place, similar to the variable `uuid_set` (line 25), is created, that will contain all storage nodes, answers from which the primary master is waiting. Transition *got\_ut* must fire if and only if this place is empty. However, it is not possible to check if the multiset is empty without inhibitor

arc. One of the solutions is to change the colour domain to **SNlist**, since a list can be checked for emptiness.

-  *wait\_ans* with colour domain **SNlist**.

A new variable **sl**: **SNlist** is also declared.


Transition *poll\_pm\_verif* is replaced by a sub-net. Here it simply takes the primary master token and puts it back. Handlers of messages, that are hidden behind them, can change the state of some variables, e.g. **uuid\_set**, and, respectively, the content of place *wait\_ans*.

Transition *ask\_ut* moves the primary master token from place *wait\_pt* to *wait\_ut*. It also sends messages to all storage nodes from place *s\_iden*. Here we see that it could be convenient to change the colour domain of *s\_iden* to **SNlist**. In this case we can directly put this list into place *wait\_ans*. Also we can write an SML function **broadcast**, that sends the same message to each node from the list.

```
fun broadcast (msgType, l) =
  List.foldr (fn (sNode, tokens) =>
    1'(msgType, sl (sNode), pm, 0) ++ tokens) [] l
```

A new variable declaration is needed: **ml**: **MESSlist**.

*Conditions on events* In this example, there is only a single transition that requires an auxiliary condition to fire. It is *ask\_ut*, since it can fire only if the partition table is operational (see lines 9, 11, 22 of figure 7). To make this check, first of all, we need to add an additional place:

-  *partition\_table* with colour domain **PT**,

together with a new variable **ptbl**: **PT**. A guard must also be added. The partition table is operational if and only if there is at least one up-to-date cell for each partition. So, we can write the following SML function.

```
1 fun operational pt = List.all (fn (_, row) =>
2   List.exists (fn (_, st) => st = UP) row) pt
```

### 3.4 Analysis and feedback

The properties the protocol should satisfy were model-checked. The outcome of this analysis was suspicious scenarios. The design approach allowed for tracing back the execution sequence in the source code, and thus the engineers could check their validity. Properties for the election of a primary master node are detailed in [4] as well as their analysis. Some 70 properties were given by the engineers (in natural language). We first grouped these properties, and figured out which ones were relevant for our purposes. Examples of such properties are “At all times, there is at most one primary master node”, which concerns only the master nodes election phase, “When the primary master is in the verification state, a partition table is selected”, which deals with the overall system. Then those properties of interest for the primary master election were analysed using several tools:



- a graphical user interface, Coloane [2];
- the CPN-AMI verification platform [3];
- CPNTools for coloured nets interface and state space analysis [1];
- the HELENA high-level nets analyser [10].

Some scenarios were due to a too coarse abstraction level, but the assumptions made during modelling did hold and guarantee the appropriate behaviour of the code. An interesting erroneous scenario pointed out the possibility of a livelock in the primary master election process. However, this never happened in practice, as the developers found out it was prevented by a side-effect of a Python function. Nevertheless, they could fix it, such a side effect being undesirable, in case it doesn't happen in a future version of Python.

The lessons learned in the Neo protocol analysis project confirmed that the modelling approach from source code made it relatively easy to trace back potentially undesirable scenarios. Of course, the level of abstraction has some influence on the results, in that some of the scenarios examined do not actually lead to an error. However, it gave the engineers a better understanding of the details of their code, of the overall process flow, and served as a basis for discussions on the details of the protocol functioning. This was particularly useful since the programmers that wrote the initial code were not available anymore.

## 4 Conclusion

In this paper we gave detailed directions on how to construct a coloured Petri net model from a high level object oriented source code and illustrated it with a real case example. The modelling process is divided into 2 main parts: grounding and code analysis.

Now the following questions could be raised. Can this process be automated? The most complicated part of the modelling process is to choose objects and actions that are important for the goals of verification and separate them from those that are not as useful. If a programming language could provide some kind of priorities to data structures and methods, it may simplify the automation process. During the industrial project in which the NEO protocol was analysed, a code-tagging approach to facilitate both the modelling and the interpretation of the verification results was envisioned for future work. Moreover, part of our group works on a tool-supported Petri net model design method from a natural language description. That tool provides the user with specific guidelines for model construction, mainly based on a refinement approach. Its integration within a larger software platform, namely COSYVERIF [15] will also provide analysis capabilities. Further work could include the integration of both approaches so as to have a better feedback on the original source code.

Another interesting question is could these modelling guidelines be applied elsewhere? Even if not directly, but with some refinements, they could be applied to any reverse-engineering process providing a coloured Petri net or a similar model of concurrency.

## References

1. CPN Tools Homepage, <http://cpntools.org/>.
2. The Coloane tool Homepage, <https://coloane.lip6.fr/>.
3. The CPN-AMI Homepage, <http://move.lip6.fr/software/CPNAMI/>.
4. C. Choppy, A. Dedova, S. Evangelista, S. Hong, K. Klai, and L. Petrucci. The NEO protocol for large-scale distributed database systems: Modelling and initial verification. In *Proc. 31st Int. Conf. Application and Theory of Petri Nets and Other Models of Concurrency (PetriNets'2010), Braga, Portugal, June 2010*, volume 6128 of *Lecture Notes in Computer Science*, pages 145–164. Springer Verlag, June 2010.
5. C. Choppy, A. Dedova, S. Evangelista, K. Klai, L. Petrucci, and S. Youcef. Modelling and formal verification of the NEO protocol. *Transactions on Petri Nets and Other Models of Concurrency*, 7400:197–225, 2012.
6. C. Choppy, L. Petrucci, and G. Reggio. A modelling approach with coloured Petri nets. In *Proc. 13th International Conference on Reliable Software Technologies / ADA-Europe, Venice, Italy, LNCS 5026*, pages 73–86. Springer-Verlag, 2008.
7. Giovanni Denaro and Mauro Pezz. Petri nets and software engineering. *Lectures on Concurrency and Petri Nets: Advances in Petri Nets*, 3098:439–466, 2004.
8. J. Desel and L. Petrucci. Aggregating views for Petri net model construction. In *Proc. workshop on Petri Nets and Distributed Systems (PNDS08, associated with Petri Nets 2008), Xi'an, China*, pages 17–31, 2008.
9. S. Duri, U. Buy, R. Devarapalli, and S. M. Shatz. Application and experimental evaluation of state space reduction methods for deadlock analysis in ada. *ACM Trans. Softw. Eng. Methodol.*, 3(4):340–380, October 1994.
10. S. Evangelista. High Level Petri Nets Analysis with Helena. In *ATPN'2005*, volume 3536 of *LNCS*, pages 455–464. Springer, 2005.
11. U. Farooq, C. P. Lam, and H. Li. Transformation methodology for UML 2.0 activity diagram into colored Petri nets. In *Proceedings of the third conference on IASTED International Conference: Advances in Computer Science and Technology, ACST'07*, pages 128–133, Anaheim, CA, USA, 2007. ACTA Press.
12. Elhillali Kerkouche, Allaoua Chaoui, El-Bay Bourennane, and Ouassila Labbani. A uml and colored petri nets integrated modeling and analysis approach using graph transformation. *Journal of Object Technology*, 9(4):25–43, 2010.
13. Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use.*, volume Basic Concepts. Springer-Verlag, 1992.
14. John Anil Saldhana and Sol M. Shatz. Uml diagrams to object petri net models: An approach for modeling and analysis. In *In International Conference on Software Engineering and Knowledge Engineering*, pages 103–110, 2000.
15. The CosyVerif group. CosyVerif Web page. <http://www.cosyverif.org>.
16. JB. Voron and F. Kordon. Transforming Sources to Petri Nets : A Way to Analyze Execution of Parallel Programs. In *International Workshop on Petri Nets Tools and Applications (PNTAP)*, pages 1–10. ACM, 2008.