Designing coloured Petri net models: a method

Christine Choppy, Laure Petrucci, and Gianna Reggio

1 LIPN, Institut Galilée - Université Paris XIII, France
2 DISI, Università di Genova, Italy

Abstract. When designing a complex system with critical requirements (e.g. for safety issues), formal models are often used for analysis prior to costly hardware/software implementation. However, writing the formal specification starting from the textual description is not easy. An approach to this problem has been developed in the context of algebraic specifications [5]. Here, we present a similar method, giving precise and detailed guidelines for writing coloured Petri nets.

Keywords: specification method, modelling method, coloured Petri net

1 Introduction

While formal specifications are well advocated when a good basis for further development is required, they remain difficult to write in general. Among the problems are the complexity of the system to be developed, and the use of a formal language. Hence, some help is required to start designing the specification, and then some guidelines are needed to remind some essential features to be described. [5] proposes a method, providing detailed and precise guidelines, for the development of specifications written using Casl [1], the Common Algebraic Specification Language, and Casl-Ltl [13], an extension for dynamic systems specification, as target languages. However, this method could be used with quite a variety of target languages.

Petri nets have been successfully used for concurrent systems specification. Among their attractive features, is the combination of a graphical language and an effective formal model that may be used for formal verification. Expressiveness of Petri nets is dramatically increased by the use of high-level/coloured Petri nets, and also by the addition of modularity features allowing for quite large case studies.

While the use of Petri nets becomes much easier with the availability of high quality environments and tools, to our knowledge, little work has been devoted to a specification method for Petri nets. The aim of this work is to provide guidelines for coloured Petri net specification on the grounds of the aforementioned specification method. An initial approach was presented in [2]. In this paper, further work is achieved in different directions. We show how the relevant items can be identified in the description. We adapted in detail the method so as to encompass the coloured Petri net target, and achieved a full treatment of properties.
It is important to mention some facts about this work. First, the example developed here (which is classical for Petri nets) was designed by the authors of the paper who are not specialists in Petri nets. Hence, they had no prior knowledge of the usual coloured net modelling the problem, and were only given a textual description of the problem. Actually, the model obtained is slightly different from the usual one. Second, they have also tried out other examples, starting with place/transition nets. The other author did validate their approach afterwards. Third, this approach was successfully used by our master students. Therefore, its application to a large category of problems seems rather promising. It can however still be refined, so as to take into account more detailed features such as hierarchy/modularity, as mentioned in our conclusions.

The paper is structured as follows. The different steps of our method (finding events and state observers, looking for properties, modelling with a coloured net, checking the properties) are explained and illustrated on a case study, in sections 3, 4, 5 and 6 respectively. Finally, section 7 concludes and indicates issues for future work.

2 The method

The goal of the proposed method is to obtain a coloured Petri net modelling a given system hereafter denoted by the System. The general approach is described in Fig. 1.

The proposed method is based on two key ingredients (or constituent features, using the terminology of [5]) that are events and state observers. Events are, as usual, something happening in the life of the System (e.g. an action of some component, or a change in some part of the System or in the value of a condition) and are considered as atomic, with zero-duration, and no two events may happen simultaneously (thus, in the case two actions are happening together, there will be a unique event). A state observer instead defines something that may be observed on the states of the System, defined by the values of some type.

A first step consists in deriving the state observers and the events characterising the System from its textual description (Sect. 3). Associated properties are then determined and expressed, leading to possible modifications of state observers and events (Sect. 4). When reaching a stable set of events, state observers and properties, the coloured net can be built (Sect. 5) and the properties checked (Sect. 6). This analysis may lead to modifications of the model, in which case the process should be repeated.
3 Identifying events and state observers

3.1 Guidelines to find events and state observers

The first task of the proposed method is then to find the events and the state observers that are relevant for the System. We propose a standard technique to perform this activity: a grammar-based analysis of an informal description of the System, as advocated by classical object-oriented methods (see e.g. [6]).

More precisely, the starting point should be a processing narrative, as used in [12], for the System, that is a text in natural language describing its behaviour. If the informal description does not completely present the behaviour of the System, but, for example, motivates the need for building the System or justifies some of its features, then it should be modified both eliminating and adding new parts.

The text should then be examined, and the verbs, the nouns (or better the verbal and the noun phrases), and the adjectives should be outlined. Unless the same words are used for different meanings, phrases are outlined only once.

This may be achieved on two copies of the text, one where the verb phrases are outlined, and the other for the noun phrases. To save space in this paper we used only one copy of our case study, using two different compatible styles, and this lead us to note that verb phrases and noun phrases can be nested.

In general, the outlined verbs (or verbal phrases) should lead to find out the events, while the outlined nouns and adjectives should lead to find out the state observers and the datatypes. The style used has an influence, e.g. the use of active/passive forms. Since events are also changes in parts of the System, and some actions may not be explicit (e.g. the water reaches the maximum level, or the engine is broken), a careful attention should be paid to verbs like “to be”, “to become”, “to reach”, etc. Thus all outlined verbs are listed, grouping together the synonyms or different phrases referring to the same concept, and each one is examined in order to decide whether it should yield an event. Each event should then be given a name (an identifier), and should be accompanied by a short sentence describing it. If two events are always simultaneous, they should be joined into a unique event.

Similarly, the outlined nouns and adjectives are listed, grouping synonyms, and examined in order to decide whether they yield datatypes or state observers.

Some potential cases are given below:

- if the noun denotes an active subpart of the System, it should not become a state observer, however it may be the case that the state of this subpart should be observed (e.g. if there is a user sending messages, check if the state of the user is relevant)
- similarly, for names of structural parts (or passive subparts) of the System (e.g. if two processes communicate by means of a channel, check if the status of the channel is relevant, for example, if it matters that it may be broken)
- the noun denotes data, it may be that it refers to some aspect of the System, and thus there should be an associated state observer.
– if the adjective refers to the System or to part of it, it should become a state observer of the form “is the adjective applicable to the System/its part?”

Each outlined state observer should then be given a name (an identifier) and a type, and should be accompanied by a short sentence describing what it observes in the System.

All the datatypes needed to type the state observers should be listed apart, together with a (chosen) name and if possible a definition or some operations.

Note 1. It may be quite helpful to group the events and the state observers, and, if these lists are not short, to add a title to these groups (e.g. events concerning the sender, or the receiver). These groupings are of course adopted in the above lists, and should be kept in further tables and formulas, so as to facilitate reading, eye-checking, and future modifications.

Note 2. To have to decide if a verbal phrase should be an event, and a noun phrase a state observer may lead to ask questions about the behaviour of the system (e.g. to decide if two actions are simultaneous).

Three lists are resulting from this step: (i) events, (ii) state observers, (iii) datatypes.

3.2 Case study: identifying events and state observers

The distributed database is a small example taken from [10] (vol. 1, pp. 21–25) which describes the communication among a set of database managers in a distributed system. The managers are supposed to keep their databases as identical as possible. Hence, each update must be followed by broadcasting the update to all the other managers, asking them to perform a similar update.

Even though this well-known example is small, it is complex enough to show how our method could help to specify it and obtain a coloured net model.

The informal description of this case study is given below with emphasis on verbal phrases, noun phrases, or both (when nested).

Informal description This example describes a very simple distributed database with n different sites (n is a positive integer, which is assumed to be greater than or equal to 2). Each site maintains its own copy of the whole database. On each site, a local database manager handles all operations.

Each manager is allowed to update its own copy of the database. Then, in order to keep subsequent consistency among all copies, it must send a message to all the other managers (so that they can perform the same update on their own copy of the database). In this example we are not interested in the actual update data.

Hence the messages sent on the network to ensure the cooperation between the different database managers require to keep track only of the header information, i.e. the sender and the receiver that are two different database managers. When a database manager makes an update, it
must then inform (by sending a message) all other \((n - 1)\) managers. Before a similar operation can take place again, all the updates should be finished. Therefore, the manager who has asked for an update has to wait until all other managers have sent back an acknowledgement. When a database manager is informed of a new update, it must achieve the corresponding update on its local copy and send back an acknowledgement.

Informal description analysis The first task to achieve is to analyse the textual description (as described in Sect. 3.1) so as to find out relevant elements about the events, the state of the system (expressed in terms of state observers), and the data involved (either directly mentioned in the text, or returned by the state observers). We first list the verb phrases and the noun phrases and discuss for each whether it leads to relevant information. Redundant texts (that describe the same thing) are grouped together. Then, the events, state observers and datatypes lists are extracted.

Verbs (verbal phrases)
- maintains its own copy \(\Rightarrow\) no event (a qualification of site)
- handles all operations \(\Rightarrow\) no event (a qualification of database manager)
- allowed to update its own copy of the database \(\Rightarrow\) no event (a qualification of database manager)
- keep subsequent consistency \(\Rightarrow\) no event (a motivation of some actions)
- for the following verbal phrases
  - send a message to all the other managers
  - inform (by sending a message) all other \((n - 1)\) managers
  - has asked for an update
- the inform EVENT is to inform that an initial update was done and that a manager has asked for an update (corresponding to the one it just did)
- perform the same update \(\Rightarrow\) the corrUpd EVENT is that a similar update (corresponding to the new one) is achieved
- to ensure the cooperation between the different database managers \(\Rightarrow\) no event (a motivation of some actions)
- keep track only of the header information, i.e. the sender and the receiver \(\Rightarrow\) this qualifies what is considered in the messages
- Before a similar operation can take place again, all the updates should be finished \(\Rightarrow\) two EVENTs here, one (allUpd) is that all updates are finished, and the other (update) is to perform an initial update (referred to by similar operation)
- wait until all other managers have sent back an acknowledgement \(\Rightarrow\) wait cannot be associated with an event
- all other managers have sent back an acknowledgement \(\Rightarrow\) recAllAck EVENT
- a database manager is informed of an initial update \(\Rightarrow\) informed EVENT
- it must achieve the corresponding update ⇒ the corrUpd EVENT is that a database manager achieves the update corresponding to the initial one made (already mentioned)
- and send back an acknowledgement ⇒ the updAck EVENT is that a local database manager sends back an acknowledgement

List of events
- inform: a database manager informs (by sending a message) all other \((n - 1)\) managers that an initial update was made
- allUpd: all updates are finished
- update: a database manager performs an initial update
- recAllAck: all other managers have sent back an acknowledgement, thus all acknowledgements are received
- informed: a database manager is informed of an initial update
- corrUpd: a database manager achieves the update corresponding to the initial one made
- updAck: a database manager sends back an acknowledgement

Note that since communication is asynchronous, to inform and to be informed are two different events. The issue whether there should be a distinction between events allUpd and recAllAck should be solved.

Nouns (noun phrases)
- distributed database with \(n\) different sites ⇒ this refers to the whole system, so it does not apply to a state observer or data
- \(n\) is a positive integer, greater than or equal to 2 ⇒ a constant value of type integer (or natural)
- sites of the distributed database and the local managers
  - site ⇒ this is a “structural part”, each site is referred to by its identifier (that is a datatype), and a question is whether its state should be characterised
  - local database manager ⇒ associated with each site, and a question is whether its state should be characterised
  - its local copy ⇒ it is managed by the local database manager
- several parts of the description mention messages, and will lead to the definition of the MESSAGE datatype
  - message . . . so that they can perform the same update ⇒ a datatype for messages requiring an update;
  - messages sent on the network ⇒ one question is whether the communication is synchronous or not, and it is decided in this case that it is asynchronous. Therefore, a state observer provides the messages sent in the network.
  - the header information, i.e. the sender and the receiver ⇒ the message datatype should include the sender and the receiver
  - acknowledgement ⇒ a datatype for another kind of messages
List of state observers
- inTransit: Set(MESSAGE) returns the messages in transit in the network

Let us note that, given the study of the noun phrases reported above, we have only one state observer at this point which is not much. More state observers will emerge from the next step when working on properties.

List of datatypes
- DBM: identities of the sites
- INT: integers, with a constant value $n$ greater than 2
- MESSAGE: messages that are either update requests or acknowledgements, and provide only the sender and the receiver of the message. At this stage, we can provide a provisional definition for this type:

  \[
  \text{MESSAGE}::= \text{Req} \left( \text{DBM}, \text{DBM} \right) \mid \text{Ack} \left( \text{DBM}, \text{DBM} \right)
  \]

4 Finding the properties

4.1 Guidelines to find the properties

Let us assume that we have the three lists (events, state observers and datatypes) produced in the previous step. Now we consider the task of finding the most relevant/characteristic properties of the System and of its behaviour, and to express them in terms of the identified events and state observers (using also the identified datatypes). Our method helps to find out these properties by providing precise guidelines (inspired by [5]) for the net designer to examine all relevant relationships among events and state observers, and all aspects of events and state observers.

The behaviour of the System can be seen as the set of all its possible “lives”, where a life is a sequence of states and events

\[
s_0 \in s_1 \in s_2 \ldots s_{n-1} \in s_n \in s_{n+1} \in s_{n+2} \ldots
\]

where each state $s_i$ defines the values of the state observers, and $s_0$ is an initial state.

For each state observer $SO$ returning a value of type $DT$ (declared as $SO: DT$), we look for:
- properties on the values returned by $SO$ (e.g. assuming $DT = \text{INT}$, $SO$ should always return positive values);
- properties relating the values observed by $SO$ with those returned by other state observers (e.g. the value returned by $SO$ is greater than the value returned by state observer $SO_1$).

For each event $EV$ we look for pre and postconditions and there may be other properties (e.g. liveness and incompatibility between events).

precondition is what must hold before $EV$ happens, i.e. a condition on the state observers such that if $s$ is a state of the System in which $EV$ happens, then this condition holds on $s$

postcondition is what must hold after $EV$ happened, i.e. a condition on the state observers such that if $s$ is a state of the System after $EV$ happened, then this condition holds on $s$
more properties  Consider a life of the System where EV happens
\( s_0 e_1 s_1 e_2 \ldots s_{n-1} e_n s_n \) EV \( s'_1 e'_1 s'_2 e'_2 \ldots \)
\( s_0 e_1 s_1 e_2 \ldots s_{n-1} e_n s_n \) is a possible past of EV, whereas \( s'_1 e'_1 s'_2 e'_2 \ldots \)
is a possible future of EV.

on the past  properties on the possible pasts of EV (e.g. the System was in
a state such that the values returned by the state observers satisfy some
condition, or a given event happened)
on the future  properties on the possible futures of EV (e.g. the System
will reach a state such that the values returned by the state observers
satisfy some condition, or a given event will happen)
vitality  when it should be possible for EV to happen (e.g. if state \( s \)
satisfies some condition, then EV may happen in \( s \), if state \( s \) satisfies some
condition, then eventually EV will happen, \( \ldots \))
incompatibility  the events EV\(_1\) such that there cannot exist a state of the
System in which both EV and EV\(_1\) may happen

Obviously, there may be some conditions fulfilled by the possible initial states
of the System. Conversely, we propose to characterise the final states when rele-
vant.

initial condition  a property about state observers that must hold in any initial
state of the System.

final condition  a property about state observers that must hold in any final
state of the System (irrelevant if the System never terminates).

While writing the properties it may happen that:
– we discover the need for operations over the datatypes, or that their defini-
tion should be made more precise and detailed \( \Rightarrow \) modify the definition of
the data types accordingly
– we need new state observers and perhaps new datatypes to express what
they observe (e.g. to express some property about an event) \( \Rightarrow \) add them
– we need new events, or an event has to be split into several other ones, or
different events turn out to be the same \( \Rightarrow \) add/split/identify the events as
required.

4.2 How to find the properties: case study

The text analysis did not bring much in terms of state observers, therefore event
properties are first expressed in natural language, and once the properties are
identified, the corresponding state observers will emerge. The only event lead-
ing to properties other than the pre/postconditions is update. When expressing
properties, we use primed notations for the value of state observers after an event
has taken place. Recall that - and + denote deletion and addition of an element
to a set or to a multiset.

Event properties
update: (a database manager d performs an initial update)
**precondition** no update is taking place (thus we introduce the state observer updating: BOOL) and d is inactive, i.e. in the inactive state (thus we introduce the state observer inactive: Set(DBM)):

\[ \text{updating} = \text{false} \land d \in \text{inactive} \]

**postcondition** d performed an update (thus we introduce the state observer updated: DBM+, where the values of datatype DBM+ are those of DBM plus None), d is not inactive anymore, and an update is taking place:

\[ \text{inactive}' = \text{inactive} - d \land \text{updated}' = d \land \text{updating}' = \text{true} \]

**more** it should always be eventually possible to make an initial update

**inform**: (a database manager d informs, by sending a message, all other managers that an initial update was performed)

**precondition** d performed an initial update:

\[ \text{updated} = d \]

**postcondition** d is waiting for the other sites to perform the subsequent updates (thus we introduce the state observer waiting: DBM+), and the messages sent to require the subsequent updates are in transit on the network. Moreover, we add to the datatype DBM an operation AllUpdReq producing all update request messages:

\[ \text{updated}' = \text{None} \land \text{waiting}' = d \land \text{inTransit}' = \text{inTransit} + \text{AllUpdReq}(d) \]

**informed**: (a database manager d is informed of an initial update)

**precondition** there is a message in transit requiring d to make a subsequent update from the site d1, and site d is inactive:

\[ \text{Req}(d1,d) \in \text{inTransit} \land d \in \text{inactive} \]

**postcondition** the request message is received by site d, i.e. it is included in the received messages, thus we introduce the state observer recMsg: Set(MESSAGE), and d is performing the required update, i.e. it is in the performing state, thus we introduce the state observer performing: Set(DBM):

\[ \text{inTransit}' = \text{inTransit} - \text{Req}(d1,d) \land \text{inactive}' = \text{inactive} - d \land \text{performing}' = \text{performing} + d \land \text{recMsg}' = \text{recMsg} + \text{Req}(d1,d) \]

**corrUpd**: (a database manager makes the update corresponding to the initial one)

The occurrence of this event corresponds to a database manager reaching the state performing, thus it is useless and we drop it from the events list.

**updAck**: (a database manager d sends back an acknowledgement)

**precondition** the database manager d performed the update, so it was in the performing state, and has received a request message from some d1:

\[ d \in \text{performing} \land \text{Req}(d1,d) \in \text{recMsg} \]

**postcondition** the database manager d is now in the inactive state, and an acknowledgement message is in transit on the network:

\[ \text{performing}' = \text{performing} - d \land \text{inactive}' = \text{inactive} + d \land \text{recMsg}' = \text{recMsg} - \text{Req}(d1,d) \land \text{inTransit}' = \text{inTransit} + \text{Ack}(d,d1) \]

**allUpd**: (all subsequent updates are finished)

This is the same as recAllAck, since an acknowledgement is sent when an update is done. Thus this event will be removed from the event list.

**recAllAck**: (all acknowledgements are received by database manager d)
**precondition** d is waiting for the acknowledgments, and all acknowledge-
ment messages are in transit on the network (we assume that they are
all received together):
waiting = d ∧ AllAcks (d) ⊆ inTransit

**postcondition** d is not waiting, the update acknowledge messages are
not in transit on the network anymore, and no update is taking place.
waiting' = undefined ∧ updating = false ∧
inTransit' = inTransit- AllAcks (d) ∧ inactive’ = inactive + d

Thus, while expressing the properties of the events, we have identified the
following new state observers:

(New) List of state observers

**inTransit:** Set(MESSAGE) returns the messages in transit on the network

**inactive:** Set(DBM) returns the set of the inactive database managers.

**updated:** DBM+ returns the database manager that did the initial update, or
None

**waiting:** DBM+ returns the database manager that is waiting, after having in-
formed the others that a subsequent update is required, or None.

**performing:** Set(DBM) returns the database managers performing the subse-
quent updates

**recMsg:** Set(MESSAGE) returns the update request messages received by the
database managers

**updating:** BOOL returns true if an update is taking place, and false otherwise.

(New) datatypes and operations over the DBM datatype

- DBM+_::= _--: DBM | None
- AllUpdReq: DBM → Set(MESSAGE)
  AllUpdReq (d)= { Req (d,d1) | d1:DBM, d ≠ d1 }
- AllAcks: DBM → Set(MESSAGE)
  AllAcks (d)= { Ack (d1,d) | d1:DBM, d ≠ d1 }

State observers properties

**inTransit:** Set(MESSAGE) (returns the messages in transit on the network)
- Requests from two different database managers are not in transit simulta-
neously:
  Req(d1, d1') ∈ inTransit ∧ Req(d2, d2') ∈ inTransit ⇒ d1 = d2
- Acknowledgements to two different database managers are not in transit simulta-
neously:
  Ack(d1, d1') ∈ inTransit ∧ Ack(d2, d2') ∈ inTransit ⇒ d1' = d2'
- There are messages in transit on the the network iff an update is taking place:
  inTransit ≠ ∅ ≡ updating = true
inactive: Set(DBM) (returns the set of the inactive database managers)
- An inactive database manager is neither waiting nor performing nor did an update:
  \[ d \in \text{inactive} \implies (d \neq \text{waiting} \land d \not\in \text{performing} \land d \neq \text{updated}) \]
- A database manager is either inactive, performing, waiting or just did an initial update:
  \[ d \in \text{inactive} \lor d \in \text{performing} \lor d = \text{waiting} \lor d = \text{updated} \]
updated: DBM+ (returns the database manager that did the initial update, or None)
- A database manager that did the initial update is neither waiting nor performing nor inactive:
  \[ \text{updated} \neq \text{None} \implies (\text{updated} \neq \text{waiting} \land \text{updated} \not\in \text{performing} \land \text{updated} \not\in \text{inactive}) \]
- If there is a database manager that did the initial update then no other one is waiting, and vice versa
  \[ \neg (\text{updated} \neq \text{None} \land \text{waiting} \neq \text{None}) \]
waiting: DBM+ (returns the database manager that is waiting, after having informed the others that a subsequent update is required, or None)
- A database manager that is waiting neither just did an initial update nor is performing nor is inactive:
  \[ \text{waiting} \neq \text{None} \implies (\text{waiting} \neq \text{updated} \land \text{waiting} \not\in \text{performing} \land \text{waiting} \not\in \text{inactive}) \]
performing: Set(DBM) (returns the database managers performing the subsequent updates)
- A performing database manager is neither waiting, nor inactive nor just did an initial update:
  \[ d \in \text{performing} \implies (d \neq \text{waiting} \land d \not\in \text{inactive} \land d \neq \text{updated}) \]
recMsg: Set(MESSAGE) (returns the update request messages received by the database managers)
- A message cannot be received and in transit on the network simultaneously:
  \[ \text{recMsg} \cap \text{inTransit} = \emptyset \]
updating: BOOL (returns true if an update is taking place, and false otherwise)
If an update is taking place, not all database managers are inactive, and if one of them is waiting then there are messages travelling on the network or received:
\[ \text{updating} = \text{true} \implies (\exists d. d \not\in \text{inactive}) \land (\text{waiting} \neq \text{None} \implies \text{inTransit} \cup \text{recMsg} \neq \emptyset) \]

initial state Initially all database managers are inactive, no update is taking place, and there is no message in transit on the network nor received
\[ (\forall d. d \in \text{inactive}) \land \text{waiting} = \text{None} \land \text{updated} = \text{None} \land \text{performing} = \emptyset) \land \text{updating} = \text{false} \land \text{inTransit} = \emptyset \land \text{recMsg} = \emptyset \]

final state There should not be any final state, since the distributed database system will never terminate.
5 Modelling using coloured Petri nets

5.1 Building the Coloured Petri Nets

At this point, we can assume that we have the list of state observers and events (plus the list of used datatypes with their operations) resulting from the previous steps, and that for each event the pre/postconditions have been expressed. Recall that we have collected also other properties about the state observers and the events, that will be checked in the last step of the method, once the net is built.

We now show how starting from the above elements, derived from the analysis of the System, we can build a coloured Petri net modelling the System itself. Obviously, the net cannot be built in all cases, so we present a canonical form for events, state observers and pre/postconditions that allow the procedure to result in a coloured net. Whenever the form is not canonical, it is possible to do some refactoring replacing the used state observers, events, and datatypes with other ones able to model the System in an equivalent way; analogously it is possible to replace the pre/postconditions with equivalent formulae. This scheme is sketched in Fig 2. We present later various patterns showing how to do the refactoring in some quite common cases.

The canonical form requires that:
1. each state observer has type $\text{MSet}(T)$ for some type $T$;
2. the pre/postconditions have the following form \(^1\)

\[
\begin{align*}
\text{pre} &\quad (\land_{i=1,...,n} \text{exp}_i \leq \text{SO}_i) \land (\land_{j=n+1,...,m} \text{exp}_j \leq \text{SO}_j) \land \text{cond}, \\
\text{post} &\quad (\land_{i=1,...,n} \text{SO}'_i = \text{SO}_i - \text{exp}_i + \text{exp}'_i) \land (\land_{j=n+1,...,m} \text{SO}'_j = \text{SO}_j - \text{exp}_j) \land \\
&\quad (\land_{h=m+1,...,r} \text{SO}'_h = \text{SO}_h + \text{exp}'_h) \land \text{cond}',
\end{align*}
\]

where
- $\text{SO}_i$ ($l = 1, \ldots, r$) are all distinct,
- the free variables occurring in $\text{exp}_i$ and $\text{exp}'_i$ ($l = 1, \ldots, r$) may occur in $\text{cond}$ and in $\text{cond}'$,
- no state observer occurs in $\text{cond}$, $\text{cond}'$, $\text{exp}_i$ and $\text{exp}'_i$ ($l = 1, \ldots, r$),
- and $\text{cond}$ and $\text{cond}'$ are first order formulae.

The pre/postconditions on event $\text{EV}$ in canonical form require that:
- before $\text{EV}$ occurs some values are contained in $\text{SO}_1, \ldots, \text{SO}_n$ and that such values are deleted and that other values are added when $\text{EV}$ occurs;
- before $\text{EV}$ occurs some values are contained in $\text{SO}_{n+1}, \ldots, \text{SO}_m$ and that such values are deleted when $\text{EV}$ occurs, but nothing is added;
- some values are added to $\text{SO}_{m+1}, \ldots, \text{SO}_r$ when $\text{EV}$ occurs.

\(^1\) $\leq$, $+$ and $-$ denote respectively the inclusion, union and the difference between multisets.
whereas \( \text{cond} \) expresses some condition over \( \text{exp}_1, \ldots, \text{exp}_m \), and \( \text{cond}' \) some condition over \( \text{exp}'_1, \ldots, \text{exp}'_r \).

Thus it will be possible to realise the behaviour of \( \text{EV} \) described by these pre/postconditions by the flowing of valued tokens throughout the places of a coloured net.

Assume that all elements are in the canonical form. The coloured Petri net is defined as follows. The state observers and the events determine the places and the transitions, while the pre/postconditions determine the arcs. Each state observer \( \text{SO} : \text{MSet}(T) \) becomes a place named \( \text{SO} \) coloured by \( T \), and each event \( \text{EV} \) becomes a transition, named \( \text{EV} \). If the pre/postconditions of an event \( \text{EV} \) have the same form as in (2), then the set of arcs is as pictured in Fig. 3.

**Petri net design patterns** The method proposed in this paper offers various patterns that may help to refactorise the state observers, the events and the pre/postconditions to reach a canonical form, and thus to be able to generate a coloured Petri net. Similar to design patterns in [8] in a specific case several patterns may be applicable. In this subsection we present the two patterns that will be applied in the case study.

**Black-Box Value Pattern** This pattern may be applied whenever we want to reflect in the Petri net that a state observer \( \text{SO} : T \) observes values considered as black-blox, that is we are not interested in exploiting the structure (if any) of the observed values, i.e. of \( T \).

Assume we have a state observer \( \text{SO} : T \) where \( \text{SO} \) appears in the pre/postconditions only in atoms having either the form \( \text{SO} = \text{exp} \) or \( \text{SO}' = \text{exp}' \), and \( \text{SO} \) does not appear in \( \text{exp} \) nor \( \text{exp}' \).\(^2\)

\(^2\) This is not restrictive at all, since a complex formula \( \text{cond} \) in which \( \text{SO} \) appears can always be transformed into an equivalent one \( \text{SO} = \text{exp} \land \text{cond}[\text{exp}/\text{SO}] \).
The logical specification of the System may be refactorised by replacing SO with SO : MSet(T), while the pre/postconditions should be transformed as follows:

- pre: SO = exp ∨ cond post: SO' = exp' ∨ cond' should become:
  pre: exp ≤ SO ∨ cond post: SO' = SO - exp + exp' ∨ cond'  
- pre: SO = exp ∨ cond post: cond' where SO does not appear, should become:
  pre: exp ≤ SO ∨ cond post: SO' = SO - exp ∨ cond'  
- pre: cond where SO does not appear, post: SO' = exp' ∨ cond' should become:
  pre: X ≤ SO ∨ cond (X having type T) post: SO' = SO - X + exp' ∨ cond'  

We can prove that if initially SO = exp, and thus SO = \{exp\}, then always size(SO) = 1 (thus we are sure that SO is correctly realised to return in each state a unique value).

**Set Value Pattern** This pattern may be applied to a state observer SO : Set(T) whenever we want the elements of the observed sets to be realised as tokens typed by T flowing in the Petri net.

Assume we have the state observer SO : Set(T), and that the pre/postcondition have the form:

pre: \((exp_1 \cup exp_2) \subseteq SO \land cond\) post: SO' = (SO - exp_2) ∪ exp_3 ∨ cond'  

where \(exp_1, exp_2\) and \(exp_3\) are sets (also empty), and SO does not appear in \(exp_1, exp_2, exp_3, cond\) and \(cond'\).

The precondition requires that SO includes the elements in \(exp_1 \cup exp_2\), whereas the occurrence of EV requires that only the elements in \(exp_2\) are removed from SO, and that the elements of \(exp_3\) are newley added to SO.

The logical specification of the System may be refactorised by replacing SO with SO : MSet(T), and transforming the pre/postconditions as follows:

pre: \((exp_1 + exp_2) \leq SO \land cond\) post: SO' = SO - (exp_1 + exp_2) + (exp_1 + exp_3) ∨ cond'  

Note that to get the canonical form \(exp_1\) should first be deleted and then added again to SO.

Unfortunately this refactoring does not guarantee that SO always returns a set, thus we should add the following property to the System, to be checked at a later stage:

\[ \neg \exists x . \{x, x\} \leq SO \] (i.e. SO is a set).

5.2 Modelling with coloured Petri nets: Case study

In Tab. 1 and 2, we summarise the events, their pre/postconditions, the state observers and the datatypes used to model the distributed database system.

The logical specification of the database system produced up to now is not in the canonical form that allows to generate a coloured Petri net, first of all because it uses state observers not typed by multisets. Hence, we now apply the patterns proposed in Section 5.1. For the sake of simplicity, we use the same
Names for the new state observers introduced by the refactoring. In Tab. 3 and 4 present the resulting events, state observers, datatypes and pre/postconditions. Note that the datatypes are unchanged.

The coloured net derived from this canonical form is presented in Fig. 4. We used the coloured Petri nets tool CPNtools [7], and its associated CPNML language. The declarations are shown in Fig. 5. The DBM+ colour set is declared DBMP as a union of database managers (man:DBM) and value None.

6 Checking the properties

6.1 Checking the properties of the System

The previous steps of our design method did exhibit several properties which must be satisfied by the System. These properties should be expressed accord-

<table>
<thead>
<tr>
<th>Events</th>
<th>State Observers</th>
<th>Datatypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>update</td>
<td>inTransit: Set(MESSAGE)</td>
<td>DBM</td>
</tr>
<tr>
<td>inform</td>
<td>inactive: Set(DBM)</td>
<td>MESSAGE ::= Req(DBM, DBM)</td>
</tr>
<tr>
<td>informed</td>
<td>updated: DBM+</td>
<td>DBM+ ::= ⊲ DBM</td>
</tr>
<tr>
<td>updAck</td>
<td>waiting: DBM+</td>
<td>BOOL ::= true</td>
</tr>
<tr>
<td>recAllAck</td>
<td>performing: Set(DBM)</td>
<td>recMsg: Set(MESSAGE)</td>
</tr>
</tbody>
</table>

Table 1. Elements of the distributed database specification

<table>
<thead>
<tr>
<th>Event</th>
<th>pre</th>
<th>post</th>
</tr>
</thead>
<tbody>
<tr>
<td>update</td>
<td>updating = false ∧ (d \in \text{inactive})</td>
<td>updating' = true ∧ inactive' = inactive − {d} ∧ updated' = d</td>
</tr>
<tr>
<td>inform</td>
<td>updated = d</td>
<td>updated' = None ∧ waiting' = d ∧ inTransit' = inTransit ∪ AllUpdReq(d)</td>
</tr>
<tr>
<td>informed</td>
<td>Req(d1, d) ∈ inTransit ∧ (d \in \text{inactive})</td>
<td>inTransit' = inTransit − {Req(d1, d)} ∧ inactive' = inactive − {d} ∧ performing' = performing ∪ {d} ∧ recMsg' = recMsg ∪ {Req(d1, d)}</td>
</tr>
<tr>
<td>updAck</td>
<td>d ∈ performing ∧ Req(d1, d) ∈ recMsg</td>
<td>performing' = performing − {d} ∧ recMsg' = recMsg − {Req(d1, d)} ∧ inactive' = inactive ∪ {d} ∧ inTransit' = inTransit ∪ Ack(d, d1)</td>
</tr>
<tr>
<td>recAllAck</td>
<td>waiting = d ∧ AllAcks(d) ⊆ inTransit</td>
<td>waiting' = None ∧ updating = false ∧ inTransit' = inTransit − AllAcks(d) ∧ inactive' = inactive ∪ {d}</td>
</tr>
</tbody>
</table>

Table 2. Pre/postconditions of the distributed database specification
<table>
<thead>
<tr>
<th>Events</th>
<th>State Observers</th>
<th>Datatypes</th>
</tr>
</thead>
<tbody>
<tr>
<td>update</td>
<td>inTransit: MSet(MESSAGE)</td>
<td>DBM</td>
</tr>
<tr>
<td>inform</td>
<td>inactive: MSet(DBM)</td>
<td>MESSAGE::=</td>
</tr>
<tr>
<td>informed</td>
<td>updated: MSet(DBM+)</td>
<td>Req(DBM, DBM)</td>
</tr>
<tr>
<td>updAck</td>
<td>waiting: MSet(DBM+)</td>
<td>DBM+:=: DBM</td>
</tr>
<tr>
<td>recAllAck</td>
<td>performing: MSet(DBM)</td>
<td>BOOL::= true</td>
</tr>
<tr>
<td>recMsg</td>
<td>updating: MSet(MESSAGE)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>inT ransit: MSet(DBM)</td>
<td></td>
</tr>
</tbody>
</table>

Table 3. Elements of the distributed database specification, in canonical form

<table>
<thead>
<tr>
<th>Event</th>
<th>pre</th>
<th>post</th>
</tr>
</thead>
<tbody>
<tr>
<td>update</td>
<td>false ≤ updating ∧</td>
<td>updating' = updating − false + true ∧</td>
</tr>
<tr>
<td></td>
<td>d ≤ inactive ∧</td>
<td>inactive' = inactive − d ∧</td>
</tr>
<tr>
<td></td>
<td>X ≤ updated</td>
<td>updated' = updated − X + d</td>
</tr>
<tr>
<td>inform</td>
<td>d ≤ updated ∧</td>
<td>updated' = updated − d + None ∧</td>
</tr>
<tr>
<td></td>
<td>X ≤ waiting</td>
<td>waiting' = waiting − X + d ∧</td>
</tr>
<tr>
<td></td>
<td>inTransit ≤ inTransit</td>
<td>inTransit' = inTransit − Req(d1, d) ∧</td>
</tr>
<tr>
<td></td>
<td>Req(d1, d) ≤ inTransit</td>
<td>inactive' = inactive − d ∧</td>
</tr>
<tr>
<td></td>
<td>d ≤ inactive</td>
<td>performing' = performing + d ∧</td>
</tr>
<tr>
<td></td>
<td></td>
<td>recMsg' = recMsg + Req(d1, d)</td>
</tr>
<tr>
<td>updAck</td>
<td>d ≤ performing ∧</td>
<td>performing' = performing − d ∧</td>
</tr>
<tr>
<td></td>
<td>Req(d1, d) ≤ recMsg</td>
<td>recMsg' = recMsg − Req(d1, d) ∧</td>
</tr>
<tr>
<td></td>
<td></td>
<td>inactive' = inactive + d ∧</td>
</tr>
<tr>
<td></td>
<td>inTransit ≤ inTransit</td>
<td>inTransit' = inTransit − AllAcks(d) ∧</td>
</tr>
<tr>
<td></td>
<td>AllAcks(d) ≤ inTransit</td>
<td>inactive' = inactive + d + d − false</td>
</tr>
<tr>
<td></td>
<td>X ≤ updating</td>
<td>updating' = updating − X + false ∧</td>
</tr>
<tr>
<td></td>
<td></td>
<td>inactive' = inactive + d</td>
</tr>
</tbody>
</table>

Table 4. Pre/postconditions of the canonical distributed database specification

In case some properties do not hold, the designer should look up for the causes of the problem by e.g., closely examining the states not satisfying the property and the paths leading to these states. This will give insight to locate the source of the problem. The model will then have to be modified accordingly, and the properties check repeated until all properties hold. It might also be the case that some properties derived from the informal specification are not correctly expressed. Then the properties should be changed and the new ones checked.
Fig. 4. Distributed database coloured Petri net

val nbdbm=3;
colset DBM=index D with 1..nbdbm;
colset DBMP=union man:DBM + None;
colset PAIRDBM = product DBM * DBM;
colset MESSAGE=union Req:PAIRDBM + Ack:PAIRDBM;
var d,d1:DBM;
var X: DBMP;
var b:BOOL;
fun AllUpdReq d = filter (fn Req(x,y) => (x<>y) andalso (x=d)| Ack(_,_) => false) (MESSAGE.all());
fun AllAcks d = filter (fn Req(_,_) => false | Ack(x,y) => (x<>y) andalso (y=d)) (MESSAGE.all());

Fig. 5. Declared types and functions for the distributed database Petri net

6.2 Checking the properties of the case study

The various properties were checked on the state space graph. For most of them, it boils down to checking that the set of graph nodes satisfying the negated property is empty. We now explain a representative excerpt of the properties, the others are given in [4], together with a picture of the analysis page.

The property of Fig. 6 is part of the state observers properties. It states that a database manager is either inactive, performing, waiting or just did an initial update:

\[ d \in \text{inactive} \lor d \in \text{performing} \lor d = \text{waiting} \lor d = \text{updated} \]

DBM.all() is the set of database managers, thus the union (denoted \(++\)) of the places (inactive, performing, updated and waiting) database manager multisets should be equal (negated by \(\text{<>}\)) to DBM.all(). Note that places waiting and updated may contain value None and this value should not be considered, and this is taken care of by function RemoveNone().
fun RemoveNone dbms = ext_col (fn man db => db) (filter (fn None => false | _ => true) dbms);

fun alldbmonce() = PredAllNodes(fn n =>
    Mark.Database'inactive 1 n ++
    Mark.Database'performing 1 n ++
    RemoveNone(Mark.Database'updated 1 n) ++
    RemoveNone(Mark.Database'waiting 1 n) <><> DBM.all());

Fig. 6. Property: state of all database managers

The following property refers to one of the properties brought up by the description analysis. There is at most one site waiting or that did an initial update (updated):

\[ \neg (\text{updated} \neq \text{None} \land \text{waiting} \neq \text{None}) \]

To check this property, we find the maximum number of tokens in places waiting and updated together, without considering the None value. This is done by checking this number for each state in the graph and taking the maximum of the previous result and the value for the current state, using the function in Fig. 7. After examining all states, the result is 1, therefore the property is satisfied.

Fig. 7. Property: only one database manager did an initial update

Finally, if an update is taking place, not all database managers are inactive, and if one of them is waiting then there are messages travelling on the network or received:

\[ \text{updating} = \text{true} \implies (\exists d \notin \text{inactive}) \land (\text{waiting} \neq \text{None} \implies \text{inTransit} \cup \text{recMsg} \neq \emptyset) \]

This property, expressed in Fig. 8 is split into two functions: upd1() to check the first part concerning the database managers, and upd2() to check the second part concerning the messages on the network.

7 Conclusion

Designing a formal specification has proved to be important to check properties of a system prior to hardware and software costly implementation. However, even if such an approach reduces both the costs and the experimenting time, designing a formal model is difficult in general for an engineer.

As mentioned in the introduction, to our knowledge little work was devoted to a specification method for Petri nets. However, we would like to mention some work done by M. Heiner and M. Heisel in [9] to combine place/transition
nets with Z specifications so as to reduce the net complexity. In [11], another approach is to rely on problem frames concepts to structure the problem before developing the Petri net.

This paper gives guidelines to help with the design process. It has proven successful with people who are not used to model with Petri nets, hence a positive point w.r.t. the applicability of the design methodology.

The main idea is to derive key features from the textual description of the problem to model, in a rather guided manner so as to deduce the important entities handled, and then to transform all this into Petri net elements. At the same time, some properties inherent to the system appear, that are also formalised and should be proven valid on the model at an early stage. When a coloured net is obtained, with these properties satisfied, further analysis can be carried out, leading to possible changes in the specification.

Our method is inspired by the one developed in [5] for simple dynamic systems specification with the CASL-LTL algebraic specification language, which also requires to look for state observers, events (or rather elementary interactions), and datatypes, but in addition provides an extensive list of potential properties one should look for. This way of handling properties has the advantage of giving ideas on the potential properties, with the drawback of systematic long lists.

While in [2], the initial approach presented kept these properties list, here we adapted the method so as to guide the search for properties in a "light" way.

In [3], we developed this method for place/transition nets with several examples. Place/transition nets could easily be used when the involved datatypes are boolean or natural numbers (and of course if the size and complexity of the problem is reasonable). Since this is a simpler case (tokens do not have a value, and the matching mechanism with the arc labels is very simple), we could develop a more systematic guidance of the specification development. In [4], we address also the issue of the choice of the appropriate family of Petri nets that may be hinted by the datatypes needed in a case study.

In this paper, we have used the classical distributed database problem as a running example so as to explain the design methodology step by step.

For this case study we present a choice for state observers that take the whole state of the system as an argument. We could have taken another option, to have
a function yielding, for any database manager site, its state, and clearly the way
to the coloured net would have been less straight. Future work will detail even
more the different ways to transform a state observer into a place.

In this work, we have stuck to commonly used datatypes, but a designer
could write his own complex types and functions to be used by his coloured net.
Reflecting them in the net is then more complex and must be done in a rigorous
way so as to ensure the applicability and the success of the approach.

Moreover, a large specification is often designed in a modular way. This is
not tackled here, but including such features, e.g. hierarchies in coloured nets,
is an important issue that we plan to address in the future. For instance, if
repeated patterns are found in the Petri net, then they can be put in subnets,
and a hierarchy may be introduced. If the net exhibits some symmetries, some
folding may occur, and the appropriate colors are introduced. An evaluation of
the method will be carried out in the near future.

Acknowledgements We thank the anonymous referees for their careful reading
and fruitful comments.

References

   In Proc. Workshop on Practical Use of Coloured Petri Nets, Aarhus, Denmark,
6. P. Coad and E. Yourdon. Object-Oriented Analysis. Prentice-Hall, Englewood
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of
   Reusable Object-Oriented Software. Addison-Wesley, 1995.
10. K. Jensen. Coloured Petri nets: basic concepts, analysis methods and practical use,
    vol. 1, vol. 2 et vol. 3. Monographs in Theoretical Computer Science, Springer-
    and Graphical Animation. In International Workshop on Advances and Applications
    of Problem Frames, 2006.
13. G. Reggio, E. Astesiano, and C. Chppy. CASL-LTL : A CASL Extension for Dy-
    person/ReggioG/ReggioEtAll03b.pdf.