Designing coloured Petri net models: a method

Christine Choppy[1], Laure Petrucci[1], and Gianna Reggio[2]

[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.

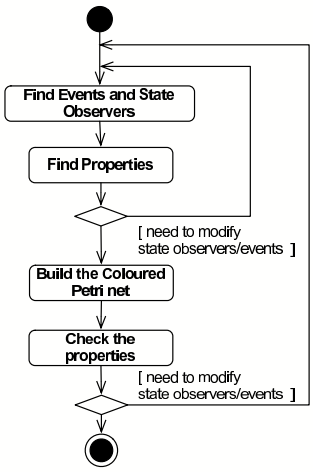

Fig. 1: Design method

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 just 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 refering 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 ackowledgement**.

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** ⇒ no event (a qualification of site)
- **handles all operations** ⇒ no event (a qualification of database manager)
- **allowed to update its own copy of the database** ⇒ no event (a qualification of database manager)
- **keep subsequent consistency** ⇒ 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** ⇒ the corrUpd EVENT is that a similar update (corresponding to the new one) is achieved
- **to ensure the cooperation between the different database managers** ⇒ no event (a motivation of some actions)
- **keep track only of the header information, i.e. the sender and the receiver** ⇒ this qualifies what is considered in the messages
- **Before a similar operation can take place again, all the updates should be finished** ⇒ 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** ⇒ **wait** cannot be associated with an event
- **all other managers have sent back an acknowledgement** ⇒ recAllAck EVENT
- **a database manager is informed of an initial update** ⇒ 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 ackowledgement** ⇒ 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:

MESSAGE::= Req (DBM,DBM) | Ack (DBM,DBM)

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

$s_0\ e_1\ s_1\ e_2\ \ldots\ s_{n-1}\ e_n\ s_n\ e_{n+1}\ s_{n+1}\ \ldots$

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 = 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, ...)

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 relevant.

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 definition should be made more precise and detailed ⇒ 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) ⇒ 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 ⇒ 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 leading 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)):
updating = false ∧ d ∈ 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:
inactive' = inactive - d ∧ updated' = d ∧ updating' = 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:
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:
updated' = None ∧ waiting' = d ∧ inTransit' = inTransit + 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:
Req (d1,d) ∈ inTransit ∧ d ∈ 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):
inTransit' = inTransit - Req (d1,d) ∧ inactive' = inactive - d ∧
performing' = performing + d ∧ recMsg' = recMsg + 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 ∈ performing ∧ Req (d1,d) ∈ recMsg

postcondition the database manager d is now in the inactive state, and an acknowledgement message is in transit on the network:
performing' = performing - d ∧ inactive' = inactive + d ∧
recMsg' = recMsg- Req (d1,d) ∧ inTransit' = inTransit + 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 form the event list.

recAllAck: (all acknowledgements are received by database manager d)

precondition d is waiting for the acknowledgments, and all acknowledgement messages are in transit on the network (we assume that they are all received together):
waiting = d $\wedge$ AllAcks (d) $\subseteq$ inTransit

postcondition d is not waiting, the update acknowlegment messages are not in transit on the network anymore, and no update is taking place.
waiting' = undefined $\wedge$ updating = false $\wedge$
inTransit' = inTransit- AllAcks (d) $\wedge$ 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 informed the others that a subsequent update is required, or None.
performing: Set(DBM) returns the database managers performing the subsequent 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 $\rightarrow$ Set(MESSAGE)
  AllUpdReq (d)= { Req (d,d1) | d1:DBM, d $\neq$ d1 }
- AllAcks: DBM $\rightarrow$ Set(MESSAGE)
  AllAcks (d)= { Ack (d1,d) | d1:DBM, d $\neq$ 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 simultaneously:
$\mathsf{Req}(\mathsf{d1}, \mathsf{d1}') \in \mathsf{inTransit} \wedge \mathsf{Req}(\mathsf{d2}, \mathsf{d2}') \in \mathsf{inTransit} \implies \mathsf{d1} = \mathsf{d2}$
– Acknowledgements to two different database managers are not in transit simultaneously:
$\mathsf{Ack}(\mathsf{d1}, \mathsf{d1}') \in \mathsf{inTransit} \wedge \mathsf{Ack}(\mathsf{d2}, \mathsf{d2}') \in \mathsf{inTransit} \implies \mathsf{d1}' = \mathsf{d2}'$
– There are messages in transit on the the network iff an update is taking place:
$\mathsf{inTransit} \neq \emptyset \equiv \mathsf{updating} = \mathsf{true}$

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

– An inactive database manager is neither waiting nor performing nor did an update:

$\mathsf{d} \in \mathsf{inactive} \implies (\mathsf{d} \neq \mathsf{waiting} \wedge \mathsf{d} \notin \mathsf{performing} \wedge \mathsf{d} \neq \mathsf{updated})$

– A database manager is either inactive, performing, waiting or just did an initial update:

$\mathsf{d} \in \mathsf{inactive} \vee \mathsf{d} \in \mathsf{performing} \vee \mathsf{d} = \mathsf{waiting} \vee \mathsf{d} = \mathsf{updated}$

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:

$\mathsf{updated} \neq \mathsf{None} \implies$
$(\mathsf{updated} \neq \mathsf{waiting} \wedge \mathsf{updated} \notin \mathsf{performing} \wedge \mathsf{updated} \notin \mathsf{inactive})$

– If there is a database manager that did the initial update then no other one is waiting, and vice versa

$\neg(\mathsf{updated} \neq \mathsf{None} \wedge \mathsf{waiting} \neq \mathsf{None})$

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:

$\mathsf{waiting} \neq \mathsf{None} \implies$
$(\mathsf{waiting} \neq \mathsf{updated} \wedge \mathsf{waiting} \notin \mathsf{performing} \wedge \mathsf{waiting} \notin \mathsf{inactive})$

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:

$\mathsf{d} \in \mathsf{performing} \implies (\mathsf{d} \neq \mathsf{waiting} \wedge \mathsf{d} \notin \mathsf{inactive} \wedge \mathsf{d} \neq \mathsf{updated})$

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:

$\mathsf{recMsg} \cap \mathsf{inTransit} = \emptyset$

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:

$\mathsf{updating} = \mathsf{true} \implies$
$(\exists \mathsf{d}.\mathsf{d} \notin \mathsf{inactive}) \wedge (\mathsf{waiting} \neq \mathsf{None} \implies \mathsf{inTransit} \cup \mathsf{recMsg} \neq \emptyset)$

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

$[(\forall \mathsf{d}.\mathsf{d} \in \mathsf{inactive}) \wedge \mathsf{waiting} = \mathsf{None} \wedge \mathsf{updated} = \mathsf{None} \wedge \mathsf{performing} = \emptyset] \wedge$
$\mathsf{updating} = \mathsf{false} \wedge \mathsf{inTransit} = \emptyset \wedge \mathsf{recMsg} = \emptyset$

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 Net

At this point, we can assume that we have the list of state observers and events (plus the list of used dataypes 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.

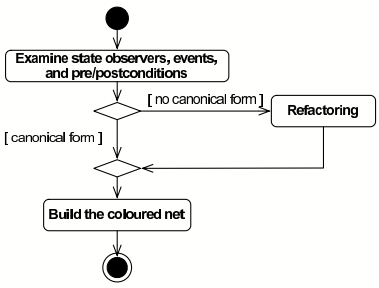

Fig. 2: Deriving the coloured net

The *canonical form* requires that:

1. each state observer has type MSet(T) for some type T;
2. the pre/postconditions have the following form [1]

$$\textbf{pre}\quad (\wedge_{i=1,\ldots,n}\ exp_i \leq \mathsf{SO}_i) \wedge (\wedge_{j=n+1,\ldots,m}\ exp_j \leq \mathsf{SO}_j) \wedge cond,$$

$$\textbf{post}\quad (\wedge_{i=1,\ldots,n}\ \mathsf{SO}'_i = \mathsf{SO}_i - exp_i + exp'_i) \wedge (\wedge_{j=n+1,\ldots,m}\ \mathsf{SO}'_j = \mathsf{SO}_j - exp_j) \wedge (\wedge_{h=m+1,\ldots,r}\ \mathsf{SO}'_h = \mathsf{SO}_h + exp'_h) \wedge cond',$$

where

- SO_l $(l = 1, \ldots, r)$ are all distinct,
- the free variables occurring in exp_l and exp'_l $(l = 1, \ldots, r)$ may occur in $cond$ and in $cond'$,
- no state observer occurs in $cond$, $cond'$, exp_l and exp'_l $(l = 1, \ldots, r)$,
- and $cond$ and $cond'$ are first order formulae.

The pre/postconditions on event EV in canonical form require that:

- before EV occurs some values are contained in $\mathsf{SO}_1, \ldots, \mathsf{SO}_n$ and that such values are deleted and that other values are added when EV occurs;

- before EV occurs some values are contained in $\mathsf{SO}_{n+1}, \ldots, \mathsf{SO}_m$ and that such values are deleted when EV occurs, but nothing is added;

- some values are added to $\mathsf{SO}_{m+1}, \ldots, \mathsf{SO}_r$ when EV occurs.

[1] $\leq$, $+$ and $-$ denote respectively the inclusion, union and the difference between multisets.

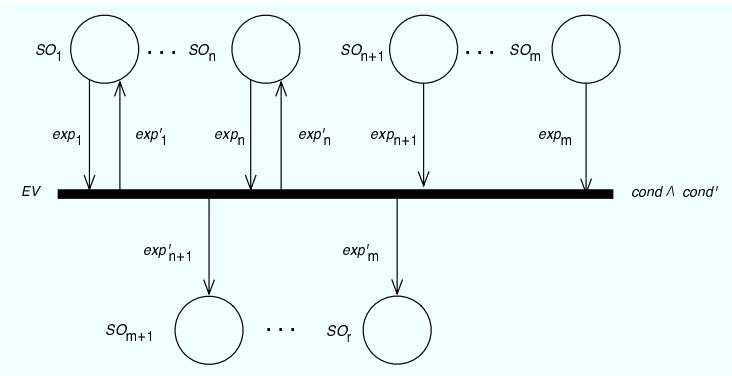

Fig. 3. Deriving arcs from pre/post-conditions

– whereas *cond* expresses some condition over $exp_1,\ldots,exp_m$, and $cond'$ some condition over $exp'_1,\ldots,exp'_r$.

Thus it will be possible to realise the behaviour of 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 $\mathsf{SO} : \mathrm{MSet}(\mathsf{T})$ becomes a place named SO coloured by T, and each event EV becomes a transition, named EV. If the pre/postconditions of an event 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 $\mathsf{SO} : \mathsf{T}$ 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 $\mathsf{SO} : \mathsf{T}$ where SO appears in the pre/postconditions only in atoms having either the form $\mathsf{SO} = exp$ or $\mathsf{SO}' = exp'$, and SO does not appear in exp nor exp'.[2]

[2] This is not restrictive at all, since a complex formula *cond* in which SO appears can always be transformed into an equivalent one $\mathsf{SO} = exp \wedge cond[exp/\mathsf{SO}]$.

The logical specification of the System may be refactorised by replacing SO with $\underline{\mathsf{SO}}$: MSet(T), while the pre/postconditions should be transformed as follows:

- pre: $\mathsf{SO} = exp \wedge cond$ post: $\mathsf{SO}' = exp' \wedge cond'$ should become:
  pre: $exp \leq \underline{\mathsf{SO}} \wedge cond$ post: $\underline{\mathsf{SO}}' = \underline{\mathsf{SO}} - exp + exp' \wedge cond'$
- pre: $\mathsf{SO} = exp \wedge cond$ post: $cond'$ where SO does not appear, should become:
  pre: $exp \leq \underline{\mathsf{SO}} \wedge cond$ post: $\underline{\mathsf{SO}}' = \underline{\mathsf{SO}} - exp \wedge cond'$
- pre: $cond$ where SO does not appear, post: $\mathsf{SO}' = exp' \wedge cond'$ should become:
  pre: $X \leq \underline{\mathsf{SO}} \wedge cond$ (X having type T) post: $\underline{\mathsf{SO}}' = \underline{\mathsf{SO}} - X + exp' \wedge cond'$

We can prove that if initially $\mathsf{SO} = exp$, and thus $\underline{\mathsf{SO}} = \{exp\}$, then always $size(\underline{\mathsf{SO}}) = 1$ (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/postcondtion have the form:

pre: $(exp_1 \cup exp_2) \subseteq \mathsf{SO} \wedge cond$ post: $\mathsf{SO}' = (\mathsf{SO} - exp_2) \cup exp_3 \wedge cond'$
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 $\underline{\mathsf{SO}}$: MSet(T), and transforming the pre/postconditions as follows:

pre: $(exp_1 + exp_2) \leq \underline{\mathsf{SO}} \wedge cond$

post: $\underline{\mathsf{SO}}' = \underline{\mathsf{SO}} - (exp_1 + exp_2) + (exp_1 + exp_3) \wedge cond'$ 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 $\underline{\mathsf{SO}}$ always returns a set, thus we should add the following property to the System, to be checked at a later stage:

$\neg\exists x \,.\, \{x, x\} \leq \underline{\mathsf{SO}}$ (i.e. $\underline{\mathsf{SO}}$ 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

Events	State Observers	Datatypes
update	inTransit: Set(MESSAGE)	DBM
inform	inactive: Set(DBM)	MESSAGE::=
		Req(DBM, DBM) | Ack(DBM, DBM)
informed	updated: DBM+	DBM+::= _: DBM | None
updAck	waiting: DBM+	BOOL::= true | false
recAllAck	performing: Set(DBM)	
	recMsg: Set(MESSAGE)	
	updating: BOOL	

Table 1. Elements of the distributed database specification

Event	pre	post
update	updating = false ∧ d ∈ inactive	updating′ = true ∧ inactive′ = inactive − {d} ∧ updated′ = d
inform	updated = d	updated′ = None ∧ waiting′ = d ∧ inTransit′ = inTransit ∪ AllUpdReq(d)
informed	Req(d1, d) ∈ inTransit ∧ d ∈ inactive	inTransit′ = inTransit − {Req(d1, d)} ∧ inactive′ = inactive − {d} ∧ performing′ = performing ∪ {d} ∧ recMsg′ = recMsg ∪ {Req(d1, d)}
updAck	d ∈ performing ∧ Req(d1, d) ∈ recMsg	performing′ = performing − {d} ∧ recMsg′ = recMsg − {Req(d1, d)} ∧ inactive′ = inactive ∪ {d} ∧ inTransit′ = inTransit ∪ Ack(d, d1)
recAllAck	waiting = d ∧ AllAcks(d) ⊆ inTransit	waiting′ = None ∧ updating = false ∧ inTransit′ = inTransit − AllAcks(d) ∧ inactive′ = inactive ∪ {d}

Table 2. Pre/postconditions of the distributed database specification

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-

Events	State Observers	Datatypes
update	inTransit: $\mathrm{MSet}(\mathsf{MESSAGE})$	DBM
inform	inactive: $\mathrm{MSet}(\mathsf{DBM})$	MESSAGE::=
		$\mathsf{Req}(\mathsf{DBM}, \mathsf{DBM}) \mid \mathsf{Ack}(\mathsf{DBM}, \mathsf{DBM})$
informed	updated: $\mathrm{MSet}(\mathsf{DBM}+)$	DBM+::= _: DBM | None
updAck	waiting: $\mathrm{MSet}(\mathsf{DBM}+)$	BOOL::= true | false
recAllAck	performing: $\mathrm{MSet}(\mathsf{DBM})$	
	recMsg: $\mathrm{MSet}(\mathsf{MESSAGE})$	
	updating: $\mathrm{MSet}(\mathsf{BOOL})$	

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

Event	pre	post
update	$\mathsf{false} \leq \mathsf{updating} \wedge$ $\mathsf{d} \leq \mathsf{inactive} \wedge$ $\mathsf{X} \leq \mathsf{updated}$	$\mathsf{updating}' = \mathsf{updating} - \mathsf{false} + \mathsf{true} \wedge$ $\mathsf{inactive}' = \mathsf{inactive} - \mathsf{d} \wedge$ $\mathsf{updated}' = \mathsf{updated} - \mathsf{X} + \mathsf{d}$
inform	$\mathsf{d} \leq \mathsf{updated} \wedge$ $\mathsf{X} \leq \mathsf{waiting}$	$\mathsf{updated}' = \mathsf{updated} - \mathsf{d} + \mathsf{None} \wedge$ $\mathsf{waiting}' = \mathsf{waiting} - \mathsf{X} + \mathsf{d} \wedge$ $\mathsf{inTransit}' = \mathsf{inTransit} + \mathsf{AllUpdReq}(\mathsf{d})$
informed	$\mathsf{Req}(\mathsf{d1}, \mathsf{d}) \leq \mathsf{inTransit} \wedge$ $\mathsf{d} \leq \mathsf{inactive}$	$\mathsf{inTransit}' = \mathsf{inTransit} - \mathsf{Req}(\mathsf{d1}, \mathsf{d}) \wedge$ $\mathsf{inactive}' = \mathsf{inactive} - \mathsf{d} \wedge$ $\mathsf{performing}' = \mathsf{performing} + \mathsf{d} \wedge$ $\mathsf{recMsg}' = \mathsf{recMsg} + \mathsf{Req}(\mathsf{d1}, \mathsf{d})$
updAck	$\mathsf{d} \leq \mathsf{performing} \wedge$ $\mathsf{Req}(\mathsf{d1}, \mathsf{d}) \leq \mathsf{recMsg}$	$\mathsf{performing}' = \mathsf{performing} - \mathsf{d} \wedge$ $\mathsf{recMsg}' = \mathsf{recMsg} - \mathsf{Req}(\mathsf{d1}, \mathsf{d}) \wedge$ $\mathsf{inactive}' = \mathsf{inactive} + \mathsf{d} \wedge$ $\mathsf{inTransit}' = \mathsf{inTransit} + \mathsf{Ack}(\mathsf{d}, \mathsf{d1})$
recAllAck	$\mathsf{d} \leq \mathsf{waiting} \wedge$ $\mathsf{AllAcks}(\mathsf{d}) \leq \mathsf{inTransit} \wedge$ $\mathsf{X} \leq \mathsf{updating}$	$\mathsf{waiting}' = \mathsf{waiting} - \mathsf{d} + \mathsf{None} \wedge$ $\mathsf{inTransit}' = \mathsf{inTransit} - \mathsf{AllAcks}(\mathsf{d}) \wedge$ $\mathsf{updating}' = \mathsf{updating} - \mathsf{X} + \mathsf{false} \wedge$ $\mathsf{inactive}' = \mathsf{inactive} + \mathsf{d}$

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

ing to the language accepted by the coloured Petri nets tool to be used. Then the properties should be checked using the tool. One possibility is to build the occurrence graph and check that all states generated satisfy the properties.

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.

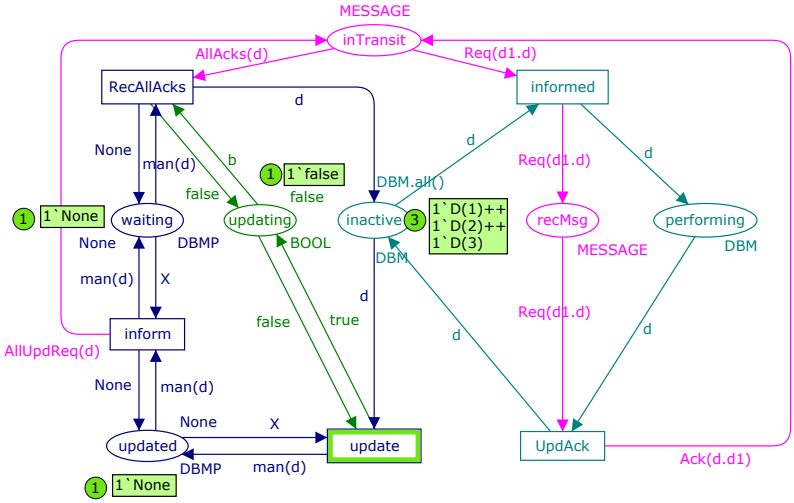

Fig. 4. Distributed database coloured Petri net

```
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*:

$\mathsf{d} \in \mathsf{inactive} \vee \mathsf{d} \in \mathsf{performing} \vee \mathsf{d} = \mathsf{waiting} \vee \mathsf{d} = \mathsf{updated}$

`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 `<><>`) 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 alldbmsonce() = 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)*:

$\neg(\mathsf{updated} \neq \mathsf{None} \wedge \mathsf{waiting} \neq \mathsf{None})$

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.

```
fun maxupdate () = SearchAllNodes(fn _ => true,
    fn n => size(RemoveNone(Mark.Database'waiting 1 n)) +
            size(RemoveNone(Mark.Database'updated 1 n)),
    0, Int.max);
```

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*:

$\mathsf{updating} = \mathsf{true} \implies (\exists \mathsf{d.d} \notin \mathsf{inactive}) \wedge (\mathsf{waiting} \neq \mathsf{None} \implies \mathsf{inTransit} \cup \mathsf{recMsg} \neq \emptyset)$

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

```
fun upd1() = PredAllNodes(fn n =>
    if (Mark.Database'updating 1 n == 1`true andalso
        Mark.Database'waiting 1 n <><> 1`None)
    then (Mark.Database'inTransit 1 n ++
          Mark.Database'recMsg 1 n == empty)
    else false);

fun upd2() = PredAllNodes(fn n =>
    if (Mark.Database'updating 1 n == 1`true)
    then (Mark.Database'inactive 1 n == DBM.all())
    else false);
```

Fig. 8. Property: an update is taking place

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

1. M. Bidoit and P. Mosses. *CASL User Manual, Introduction to Using the Common Algebraic Specification Language.* LNCS 2900. Springer-Verlag, 2004.
2. C. Choppy and L. Petrucci. Towards a methodology for modelling with Petri nets. In *Proc. Workshop on Practical Use of Coloured Petri Nets, Aarhus, Denmark*, pages 39–56, Oct. 2004. Report DAIMI-PB 570, Aarhus, DK.
3. C. Choppy, L. Petrucci, and G. Reggio. A method for modelling with place/transitions nets. Technical report, Université Paris 13, 2006.
4. C. Choppy, L. Petrucci, and G. Reggio. A method for modelling with coloured nets. Technical report, Université Paris 13, 2007.
5. C. Choppy and G. Reggio. A formally grounded software specification method. *Journal of Logic and Algebraic Programming*, 67(1-2):52–86, 2006.
6. P. Coad and E. Yourdon. *Object-Oriented Analysis.* Prentice-Hall, Englewood Cliffs, N.J., 1991.
7. The CPN Tools Homepage. http://www.daimi.au.dk/CPNtools.
8. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
9. M. Heiner and M. Heisel. Modelling Safety-Critical Systems with Z and Petri Nets. In *Proc. SafeComp '99*, LNCS 1698, pages 361 – 374. Springer-Verlag, 1999.
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-Verlag, London, UK, 1995.
11. J. Jorgensen. Addressing Problem Frame Concerns Using Coloured Petri Nets and Graphical Animation. In *International Workshop on Advances and Applications of Problem Frames*, 2006.
12. R. S. Pressman. *Software Engineering: A Practitioner's Approach, 6th edition.* McGraw-Hill, 2005.
13. G. Reggio, E. Astesiano, and C. Choppy. CASL-LTL : A CASL Extension for Dynamic Reactive Systems Version 1.0– Summary. Technical Report DISI-TR-03-36, DISI – Università di Genova, Italy, 2003. Available at `ftp://ftp.disi.unige.it/person/ReggioG/ReggioEtAll03b.pdf`.