A Modelling Approach with Coloured Petri Nets

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 [CR06], and was later adapted to Petri nets [CP04, CPR07]. Here, we show how such a method, with precise and detailed guidelines, can be applied for writing modular coloured Petri nets. This is illustrated on a model railway case study, where modules are a key aspect.

Keywords: specification method, modelling method, coloured Petri nets, modular design.

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 essential features to be described.

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 Petri nets [JKW07], and also by the addition of modularity 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 had been devoted to a specification method for writing Petri nets.

Inspired by the work on algebraic specifications in [CR06], we proposed a method, providing detailed and precise guidelines. An initial approach was presented in [CP04], and further developed in [CPR07] where the different steps for building a coloured Petri net from a textual description of a system are shown.

In this paper, we push our work a step further and start introducing the use of modularity. Section 2 gives an overview of our design method. The role of the different steps is explained. In the following sections, these steps are detailed individually before being applied to a model railway case study. First, section 3 describes the running example and its expected behaviour in an informal way, as could be given to a designer. The *constituent features* and *modular structure* of our system are derived from the description. Section 4 expresses the expected properties of the system in terms of the previously identified elements. Then, sections 5 and 6 show how this is all transformed into a modular coloured Petri net and the properties validated. In these different steps, the basic operations from [CPR07] are summarised while focusing on the new modular aspects. In particular, we shall see that sometimes it is sufficient to consider modules independently of one another, whereas for other issues it is necessary to consider the system as a whole. Finally, section 7 discusses re-engineering (because some properties were not valid). This re-engineering phase modifies part of the train routing policy in some modules identified during the properties verification phase. The conclusion (section 8) summarises the design method and draws lessons from this experience w.r.t. modularity, refinement and re-engineering.

2 Overview of the Design Method

The goal of the proposed method is to obtain a modular coloured Petri net modelling a given system. The general approach is described in Fig. 2. While a modular structure is being built, the method is based on two key ingredients, the *constituent features*, that are *events* and *state observers*. *Events* are, as usual, e.g. an action of some component, or a change in some part of the system. A *state observer* instead defines something that may be observed on the states of the system, defined by the values of some type. These constituent features are grouped into the relevant *modules*, that represent the different components (or subsystems) of the system. Both events and state observers can *appear in different modules* when they are part of their interface (e.g. synchronised events and shared resources).

Starting from an informal (textual) description, the first step consists in identifying the state observers and the events characterising the system, as well as the components in which they take part. This leads to a set of modules, each with two lists of events and state observers: those that are proper (local) to the module, and those that are part of its interface. The identified data types may also be local or global,

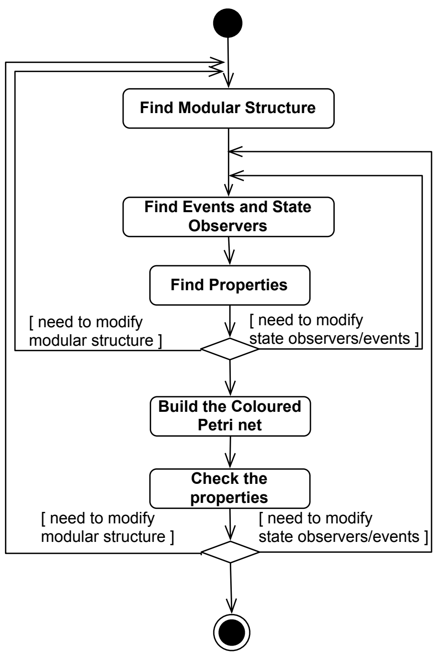

Find Modular Structure
Find Events and State Observers
Find Properties
[need to modify modular structure]
[need to modify state observers/events]
Build the Coloured Petri net
Check the properties
[need to modify modular structure]
[need to modify state observers/events]

Fig. 1. Design method

depending on whether they are used
by a single module or several. Note that in Figure 2, the identification of modules and constituent features are separated. This is due to the re-engineering phase, which may lead to modifications at a more or less large extent.

Associated properties are then determined and expressed, leading to possible modifications of state observers and events. New ones can be introduced and conversely others may be removed if they are duplicates. The lists of identified elements are updated accordingly.

When reaching a stable set of events, state observers, datatypes and properties, the modular coloured Petri net can be built and the properties checked. Several modular constructs for Petri nets exist, which basically correspond to place and transition fusion. We chose to adopt the presentation of [Kin07] since it has a clear presentation of both modules and their interface, and is considered as a good candidate within the Petri nets extensions standardisation process (ISO/IEC 15909-3 [Pet07]).

The analysis may lead to modifications of the model, in which case the process should be repeated. The nature of the modifications may be within the modules, or involve large parts of the system and therefore require reconsidering the modular structure.

In the following sections, we describe shortly the different steps and apply them to a model railway case study. For more details about the individual steps in a non-modular framework, the reader is referred to [CPR07].

3 Analysing the System Description

3.1 Guidelines for Identifying Modules and Constituent Features

The first task of the proposed method is to find the events and the state observers that are relevant. A grammar-based analysis of an informal description is proposed, as advocated by classical object-oriented methods (see e.g. [CY91]). Some figures may be part of this description, and be refered to and/or (partly) commented in the text.

The text describing the system is examined, and the verbs, the nouns (or better the verbal and the noun phrases), and the adjectives outlined. Unless the same words are used for different meanings, phrases are outlined only once. Note that verb phrases and noun phrases can be nested. There may also be sentences that do not carry any information, and are therefore discarded.

In general, the outlined verbs (or verbal phrases) lead to find out the events, while the outlined nouns and adjectives lead to find out the modules, the state observers and the datatypes.

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 is then given a name (an identifier), accompanied by a short sentence describing it. Similarly, the outlined nouns and adjectives are listed, grouping synonyms, and examined in order to decide whether they yield modules, datatypes or state observers. Each outlined state observer is then given a name (an identifier) and a type, 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.

The picture and the textual description can lead to identify *modules*, either because a particular complex entity is mentioned (e.g. a sender and a receiver in a network protocol) or because it becomes obvious that some of the other elements are strongly related to each other. In this latter case, these elements should be grouped together within a same module. It might also be the case that this module structure does not appear at this stage, but later on.

When modules are identified, they contain state observers and/or events. They are also linked to other modules in the global system, through an *interface*. The elements of the interface can be state observers or events participating in several modules whereas the other ones are local to the sole module they are involved in. As concerns datatypes, they can either be particular to a single module (e.g. a characteristic of a sender process) and can be declared locally, or shared by several modules (e.g. a message type) in which case we shall consider them as global.

For the system and each module, three lists are resulting from this step: (i) events, (ii) state observers, (iii) datatypes.

3.2 Case Study: Identifying Events and State Observers

The running example is a model railway issued from [BP01], where it was used as a case study for a students' project. 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. The model railway is depicted in Figure 2. It consists of *about 15 meters of tracks, divided into 12 sections (blocks B1 to B12) connected by four switches.* The way **the trains can pass the switches** is indicated by the arrows in Figure 2. **The traffic on all tracks can go both ways**. *The railway is connected to a computer via a serial port which allows to read information from sensors and send orders to trains through the tracks or directly to switches. Each section is equipped with one sensor at each end,* ***to detect the entrance or***

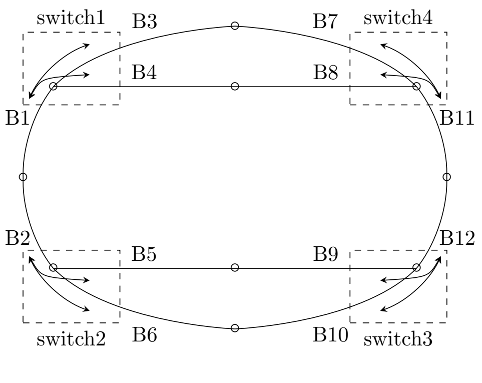


Fig. 2. The tracks of the model railway

***exit of a train*. The orders sent to trains can be either stop or go forward/backwards at a given speed**.

This railway is used by a toy shop to attract people in the store. Hence the company wants to **run several trains at the same time**, but these should *not be subject to accidents (i.e. there should be no collision)* and should **run forever without human intervention**. Thus, an adaptive routing strategy will be adopted: **the behaviour of trains adapts to local conditions instead of following a pre-determined route**. Namely, at each switch, **the train route can be chosen among several tracks and a train may even go back when it cannot continue forward**.

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 *subsystems* (or modules), 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).

Here we also have to deal with a picture, and parts of it are commented in the text, but not necessarily all. For instance, the picture clearly suggests the switches as subparts, which may yield some `Switch` *modules* in a hierarchy. Note that, in the picture, all switches are alike, i.e. there is one track on one side and two on the other. The arrows also indicate that a train goes from one side of the switch to the other side (e.g. from `B1` to `B3`, and not from `B3` to `B4`).

In this analysis of the text, we discard sentences that do not bring any elements for our concern, e.g. :
"The model railway is depicted in Figure 2."
"This railway is used by a toy shop to attract people in the store."

The text is not fully a "processing narrative", different levels of discourse are mixed, such as the "physical (or hardware) level" (e.g. describing how the railway is connected to a computer) which is not relevant for us, and the "logical level" which provides us with the needed information. There are also some slightly ambiguous parts, e.g. the "speed" of trains is mentioned but from the context we can assume it is expected to be constant except when the train stops. It is also mentioned that a train can stop, however, again from the context of the shop we understand that the trains are not supposed to remain stopped.

We also choose the category (verbal or nominal phrase) depending on the kind of information we expect to extract, for instance in the sentence:
"It consists of *about 15 meters of tracks, divided into 12 sections (blocks B1 to B12), connected by four switches.* ", "it consists" is not a verb potentially related to any event. We rather have a description of the (here permanent) state of the tracks display. Sentences about "sensors" are usually related with state observers, while sentences about "orders" may be related with some events.

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 modules, events, state observers and datatypes lists are extracted.

Verbs (verbal phrases)

– Several sentences refer to the moves and the positions of the trains:
  - **the trains can pass the switches**
  - **The traffic on all tracks can go both ways**
  - **a train may even go back when it cannot continue forward**.
  - **The orders sent to trains can be either stop or go forward/backwards at a given speed**.

  ⇒ the changeTrackSec event expresses that a train is moving from one track section to another, and the changeTrackSwitch event expresses that a train is moving from one track section to another through a switch. As mentioned above, an order is often associated with some events that are induced by the order "completion". Some event(s) may be associated with a train passing a switch (specified, e.g. by its number). It is also mentioned that trains can go "both ways", this information can be expressed either by a state parameter of a train (forward/backward/stopped), and/or by stating the tracks from and to which a train is travelling. The "speed" does not need to be a state parameter of a train.

– **run several trains at the same time** ⇒ here is an example of a verbal phrase that may refer to a state of the system where several trains are travelling at the same time, and this may be observed

– **run forever without human intervention** ⇒ rather a property of the system that does not reach a final state

– ⇒ The two sentences below express a non-determinism property
  - **the behaviour of trains adapts to local conditions instead of following a pre-determined route**.
  - **the train route can be chosen among several tracks and ...**

List of events

There are several events of two kinds:

changeTrackSec. A train is moving from one track section to another (not via a switch), for instance from `B1` to `B2`, and from `B2` to `B1`, ...

changeTrackSwitch. A train is moving from one track section to another through a switch, e.g. for Switch1, from `B1` to `B3`, from `B1` to `B4`, from `B4` to `B1`, and from `B3` to `B1`.

Nouns (noun phrases)

– *about 15 meters of tracks, divided into 12 sections (blocks B1 to B12), connected by four switches.* ⇒ as mentioned above, this refers to the permanent state of the tracks display, and this sentence does not describe in detail which track is connected to which others, and whether it is a simple connection or via a switch, since all this information is shown in Figure 2; often, state observers relate to some chosen information (rather than to the whole state), and further work (on the properties) will point out which are needed. However, it should be appropriate to have the TrackSection datatype. As mentioned earlier, we have Switch modules. Similarly, we can have modules associated with the (non-switch) connections between sections.

- *The railway is connected to a computer via a serial port which allows to read information from sensors and send orders to trains through the tracks or directly to switches.* ⇒ this describes the electronic part of the system which is not considered in the specification.
- *Each section is equipped with one sensor at each end,* ***to detect the entrance or exit of a train****.* ⇒ a sensor typically is an observer, trainPresent observes whether a train is present on a given track section (or not), thus the Train datatype provides either a train identifier or "none" that denotes that there is no train.
- *not be subject to accidents (i.e. there should be no collision)* ⇒ this is a property that should be ensured by the system.

List of datatypes

TrainId ::= $\{t_1, \dots, t_n\}$ where n is the number of trains.
TrackSection ::= `B1` | `B2` | ... `B12`   **Train** ::= TrainId | none

List of state observers

trainPresent : TrackSection → Train
observes whether a train is present on a given track section (or not)

List of modules

System is the (toplevel) module of the whole system.
Switch1, Switch2, Switch3, Switch4 are the four modules associated with the four switches, where the details of the train moves within a switch are expressed. As noted before, they are all alike and can be instantiations of a same module **Switch**. Each switch is connected to one track section on one side (that we shall name `O` afterwards) and two on the other (`T1` and `T2`). The corresponding event names are constructed with the name of the initial track and the name of the destination track, e.g. `OT1` means a move from `O` to `T1`.
MoveSec1, ... MoveSec6 similarly, we can introduce details of the train moves from one section to another one simply connected (no switch) in both ways, as instantiations of a module **MoveSec** (with sections `S1` and `S2`).

In table 1, we summarise the elements identified for each module. Those in the **System** are global for the system and thus inherited by the other modules. The other modules have a local part and an interface.

Table 1. Events, state observers and datatypes per module

System	Switch	MoveSec
TrackSection	local: changeTrackSwitch	local: changeTrackSec
Train, TrainId	`OT1`, `OT2`, `T1O`, `T2O`	`S1S2`, `S2S1`
trainPresent	interface: `O`, `T1`, `T2`: Train	interface: `S1`, `S2`: Train

4 Expected 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 for the net designer to examine all relevant relationships among events and state observers, and all aspects of events and state observers.

4.1 Finding Properties

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).

The state observers also allow for expressing the following properties:

- *initial condition*: a property about state observers that must hold in any initial state;
- *final condition*: a property about state observers that must hold in all final states, if any.

For each event EV we look for its:

- *precondition* which must hold before EV happens;
- *postcondition* which must hold after EV happened;
- other properties:
  - *on the past* : properties on the possible pasts of EV;
  - *on the future* : properties on the possible futures of EV;
  - *vitality* when it should be possible for EV to happen;
  - *incompatibility*: the events EV_i such that there cannot exist a state in which both EV and EV_i may happen.

While writing the properties, we may have to revise the lists obtained at the previous step, either to add new elements or to remove duplicates.

4.2 Properties of the Model Railway Case Study

Event properties

changeTrackSec a train tr is moving from one track section ts1 to another ts2

precondition the two tracks should be connected (we introduce a new state observer connected), there should be no train on ts2, and the train is on ts1 and is moving in the direction of ts2 (in the given layout of Figure 2 a simple and generic way to denote the train direction td is clockwise and

anticlockwise that are the two values of a TrainDirection type, and the TrainId should now include this information together with an operation direction to retrieve it ; moreover, the connected observer should include this parameter)
connected (ts1, ts2, td) ∧ direction (tr)=td ∧ trainPresent (ts1)= tr ∧ trainPresent (ts2)=none

postcondition the train is on ts2, and there is no train on ts1 anymore
trainPresent (ts2)=tr ∧ trainPresent (ts1)=none

more incompatibility properties (it is not possible that several events occur concurrently towards the same track).

changeTrackSwitch a train tr is moving from one track section ts1 to another ts2 through a switch

precondition the two tracks should be connected via a switch (we introduce a new state observer switched), there should be no train on ts2, and the train should be moving in the direction of ts2
switched (ts1, ts2, td) ∧ direction (tr)=td ∧ trainPresent (ts1)= tr ∧ trainPresent (ts2)=none

postcondition the train is on ts2, and there is no train on ts1 anymore
trainPresent (ts2)=tr ∧ trainPresent (ts1)=none

more incompatibility properties (it is not possible that several events occur concurrently towards the same track).

While expressing the properties of the events, we identified the following new state observers, datatypes and operations:

(New) List of state observers

connected : TrackSection × TrackSection × TrainDirection → BOOL
switched : TrackSection × TrackSection × TrainDirection → BOOL

(New) datatypes and operations over the TrainId datatype

TrainNumber ::= $\{t_1, \ldots, t_n\}$ where n is the number of trains.
TrainDirection ::= clockwise | anticlockwise
TrainId ::= pair (TrainNumber,TrainDirection)
direction: TrainId → TrainDirection
direction (pair (tn,td))=td
Train ::= TrainId | none (unchanged)

State observers properties

trainPresent : TrackSection → Train
observes whether a train is present on a given track section (or not), and this depends on the state of the system

connected : TrackSection × TrackSection × TrainDirection → BOOL
these are axioms about the layout, e.g.
connected (B2, B1, clockwise)=true; connected (B2, B4, anticlockwise)=false;

switched : TrackSection × TrackSection × TrainDirection → BOOL
these are axioms about the layout, e.g.
switched (B3,B1,anticlockwise)=true; switched (B3,B5,clockwise)=false;

initial state. Initially, n trains are on different tracks, each heading one direction or the other.
$\forall \mathsf{tr} \in \mathsf{TrainNumber} : \exists! \mathsf{ts} \in \mathsf{TrackSection} : \exists \mathsf{d} \in \mathsf{TrainDirection} :$
$\mathsf{trainPresent}(\mathsf{ts}) = (\mathsf{tr}, \mathsf{d})$

final state. There should not be any final state, since the system should never terminate.

Note that switches 1 and 3 behave identically since a train present on `O` heading clockwise can go on either `T1` or `T2`, while it is the case in switches 2 and 4 if the train is running anticlockwise. Therefore, the `Switch` module can be parameterised with the direction (as in section 5.2). This entails that for a module `Switch(dir)`, we have: switched (`O`,`T1`,`dir`)=true; switched (`O`,`T2`,`dir`)=true; switched (`T1`,`O`,`!dir`)=true; switched (`T2`,`O`,`!dir`)=true, where `!dir` is the direction opposite to `dir`, and all the other possibilities are false. A similar approach can be applied to connected in the `Move Sec` modules.

5 Construction of the Modular Coloured Petri Net

At this point, we can assume that we have the list of modules with their interfaces, as well as the 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. Other properties about the state observers and the events have also been found, that will be checked in the last step of the method, once the net is built.

5.1 Deriving the Net

Starting from the above elements, a coloured Petri net modelling the system can be built from the different modules and their interfaces. For each module, we first express the conditions in a canonical way. 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]
   pre $(\wedge_{i=1,\ldots,n}\, exp_i \leq \mathsf{SO}_i) \wedge (\wedge_{j=n+1,\ldots,m}\, exp_j \leq \mathsf{SO}_j) \wedge cond,$
   post $(\wedge_{i=1,\ldots,n}\, \mathsf{SO}'_i = \mathsf{SO}_i - exp_i + exp'_i) \wedge (\wedge_{j=n+1,\ldots,m}\, \mathsf{SO}'_j = \mathsf{SO}_j - exp_j) \wedge$
   $(\wedge_{h=m+1,\ldots,r}\, \mathsf{SO}'_h = \mathsf{SO}_h + exp'_h) \wedge cond',$
   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.

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

In [CPR07], some often encountered schemes have been identified so as to obtain a canonical form.

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 SO : MSet(T) becomes a place named SO coloured by T, and each event EV becomes a transition, named EV. Pre/postconditions of an event EV lead to the set of arcs as pictured in Fig. 3.

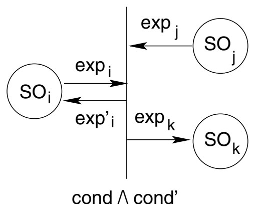


Fig. 3. Deriving arcs

5.2 Coloured Petri Net Modelling the Railway

We deduce from the previous analysis that the net modelling the railway is composed of 4 `Switch` (figure 5(b)) and 6 `MoveSec` modules (figure 5(a)). Moreover, a *toplevel* structure (figure 4) indicates how these different modules are linked together via their interfaces. The notations adopted here are those of [Kin07]. For the sake of figures readability, anticlockwise and clockwise are shortened to `acl` and `cl` respectively. Note that the track sections are modelled by places containing a token with the contents of the section itself.

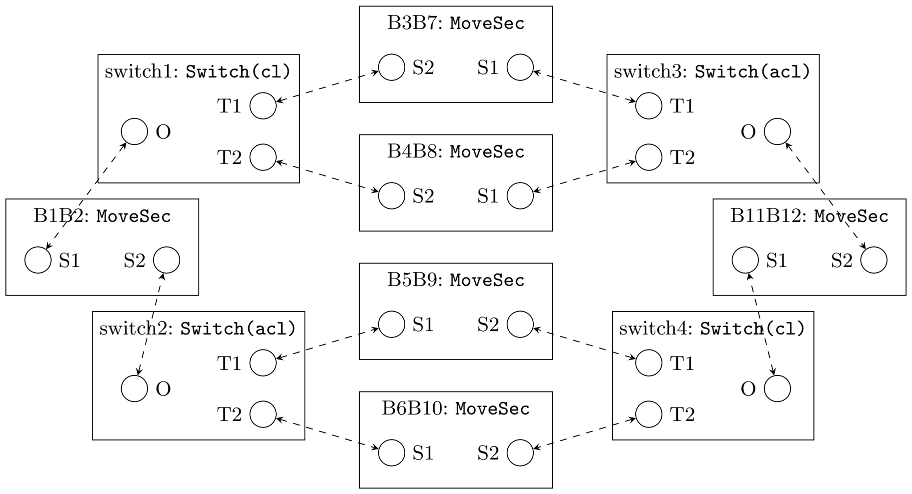
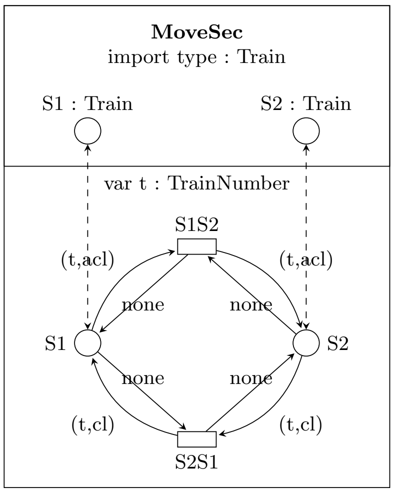


Fig. 4. The toplevel net model of the model railway

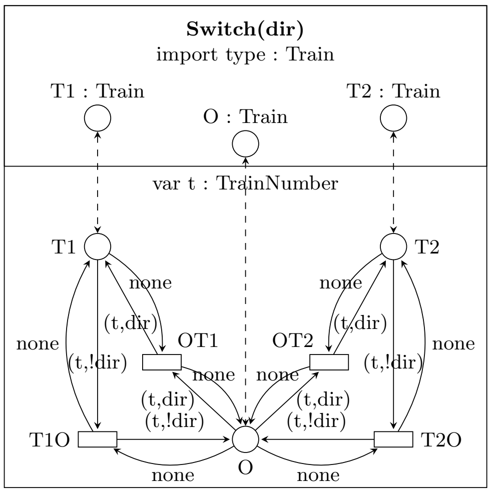
6 Checking the Properties

6.1 Checking the Expected and Required Properties

The previous steps of our design method did exhibit several properties which must be satisfied by the system. These properties should be expressed according to the language accepted by the coloured Petri nets tool to be used. Then the



(a) The `MoveSec` module



(b) The `Switch(dir)` module

Fig. 5. The modules

properties should be checked using the tool, e.g. generate the occurrence graph and check that all states satisfy the properties.

6.2 The Railway Properties

After generating the occurrence graph for an initial marking with 4 trains, we check the expected properties. We find that there are deadlocks, e.g. with a train going clockwise in B11 and a train going in the other direction in B12. The other deadlocks are similar with either adjacent track sections or sections connected by a switch. Hence the policies for moving between adjacent tracks must be improved in both kinds of modules.

7 Re-engineering

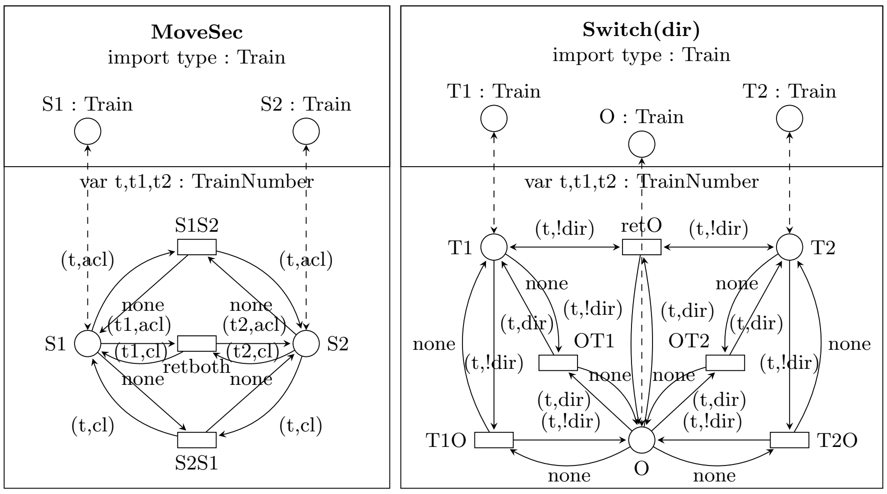
7.1 Modifying the Model

In case some properties do not hold, the designer should investigate the causes of the problem by e.g. closely examining the states not satisfying the property and the paths leading to these states. This gives insight to locate the source of the problem. The model then has to be modified accordingly, and the process 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.

7.2 New Version of the Railway Model

The policy in module `MoveSec` is changed by having a new event `retboth` where both trains return when each of them wants to go on the other train track, as depicted in figure 6(a). Similarly, the policy in module `Switch` is improved by adding an event retO where in case of deadlock, the train on the side of the switch with a single section returns to go in the opposite direction as in figure 6(b).

The new model is analysed again and the properties are satisfied.



(a) The new `MoveSec` module (b) The new `Switch(dir)` module

Fig. 6. The new modules

8 Conclusion and Future Work

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.

This paper gives guidelines to help with the design process. 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, inspired by [CR06], was developed in [CPR07] for writing flat coloured Petri nets. Here, we have started exploring the addition of a modular structure, which is most helpful when designing large systems. The process is applied to a simple model railway case study, which nevertheless raises issues for future work.

The process for obtaining modules should be investigated further and formalised. In the case study, the `Switch` module emerged early in the specification process. On the contrary, it seemed relevant and consistent to introduce `MoveSec`.

The notations used for the description of modules are those of [Kin07]. However, it does not completely take into account the main (toplevel) system description. Therefore, a clean expression of the hierarchy and the connection between components interfaces is required.

When listing the constituent features of the modules, some were obviously part of a single module, hence local, while others were shared by several modules. This is particularly the case for datatypes. We chose here to make these latter global by declaring them at the system level. However, for efficiency purposes, in particular during the analysis phase, we should rather consider which modules use them and which ones do not.

Our case study did exhibit several instances of a same module, and then a parameterised one. Here, finding these elements was rather simple, but we should investigate different cases where the use of such concepts is worthwhile.

The last phase of our method aims at checking that the system model satisfies the expected properties. However, we could imagine adding some refinement procedure there, in order to describe part of the system with additional detail.

Finally, the verification was performed using CPNTools [JKW07]. For the moment, no tool suite handles these modular mechanisms, having an interface to modules with possibly both places and transitions. The development of modular nets in the framework of ISO/IEC 15909-3 standardisation will not only enhance the theoretical constructs and notations, but also be an incentive for adequate tool implementation.

References

[BP01] Berthelot, G., Petrucci, L.: Specification and validation of a concurrent system: An educational project. Journal of Software Tools for Technology Transfer 3(4), 372–381 (2001)

[CP04] Choppy, C., Petrucci, L.: Towards a methodology for modelling with Petri nets. In: Proc. Workshop on Practical Use of Coloured Petri Nets, Aarhus, Denmark, October 2004, pp. 39–56 (2004) Report DAIMI-PB 570, Aarhus, DK

[CPR07] Choppy, C., Petrucci, L., Reggio, G.: Designing coloured Petri net models: a method. In: Proc. Workshop on Practical Use of Coloured Petri Nets, Aarhus, Denmark (October 2007)

[CR06] Choppy, C., Reggio, G.: A formally grounded software specification method. Journal of Logic and Algebraic Programming 67(1-2), 52–86 (2006)

[CY91] Coad, P., Yourdon, E.: Object-Oriented Analysis. Prentice-Hall, Englewood Cliffs (1991)

[JKW07] Jensen, K., Kristensen, L., Wells, L.: Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. Journal of Software Tools for Technology Transfer 9(3-4), 213–254 (2007)

[Kin07] Kindler, E.: Modular PNML revisited: Some ideas for strict typing. In: Proc. AWPN 2007, Koblenz, Germany (September 2007)

[Pet07] Petrucci, L.: ISO/IEC 15909 — Part 3: Extensions (November 2007) Working document of ISO/IEC JTC1-SC7-WG19, ref. PA2-018