Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Specification and Validation of a Concurrent System: An Educational Project

Gérard Berthelot[1], **Laure Petrucci**[2]

[1] CEDRIC-IIE, Institut d'Informatique d'Entreprise,
18 allée Jean Rostand, F-91025 EVRY Cedex
e-mail: `berthelot@iie.cnam.fr`

[2] LSV, CNRS UMR 8643, ENS de Cachan
61 avenue du président Wilson, F-94235 CACHAN Cedex
e-mail: `petrucci@lsv.ens-cachan.fr`

Received: date / Revised version: date

Abstract. For several years we have been in charge of a course on specification and validation of concurrent and reactive systems. At the end of this course, the students must carry out a project dealing with a model railway. They have to specify the railway, to validate their model, finally translate it into a program controlling the model railway with up to five trains.
In this paper, after presenting the project, we describe how the railway is specified and checked, step by step, by the students. We also explain how the analysis results lead to a policy for the switch control. Finally, we include some remarks about the implementation.

1 Presentation of the Problem

Several experiences of students modeling a toy railway have been reported, for instance [13, 7], but only a few include formal specification and even fewer strict verification. Other papers deal with specification of real train systems using Petri nets, either in order to compute time schedule [4] or in order to obtain a very detailed specification which eliminates the possibility of occurrence graph analysis [14, 8]. Also, teaching experiences with coloured Petri nets have been described in [3] which make intensive use of simulations and not of formal verification. Some papers [1, 2] are dealing with the satisfaction of railway requirements, that should be valid for any reachable marking (also satisfying these requirements), with a limited number of trains.

In this paper, we report about our teaching experience with a group of twenty graduate students during their second year in an engineering school. Their background includes two years of studies of mathematics and physics, followed by one year of computer science. During the preceding year thestudents learned the basics of computer programming, operating systems (in particular Unix processes), and concurrent programming concepts. Our course (80 hours of lectures and exercises) is optional and is composed of several topics: communicating automata, coloured Petri nets, temporal logics, real-time systems and tools for specification and verification (Design/CPN [11], Spin [12], Esterel [6]). About 20 hours are devoted to coloured Petri nets and Design/CPN. Afterwards, the students have to carry out a project by themselves. They spend roughly 70 hours to complete the project from specification to hardware control. The students choose one of the three indicated tools. In this paper, we will focus on the use of Design/CPN only.

1.1 Goals of the Project

Since the students are used to programming in a sequential style, they have difficulties mastering a software in which several concurrent entities are interacting. In order to overcome this difficulty, we decided to have them manage a system with intrinsic parallelism. A model railway, allowing several trains, is altogether well-known, convenient, inexpensive, and appealing to students. Moreover, such a system can be used to illustrate real-time concepts and tools. Therefore, we decided to create a course including all aspects of parallel programming, from the specification phase to the real-time programming and implementation. However, this course is still being developed and we have not yet addressed temporal aspects. This can be done in the future using, for instance, interval timed coloured Petri nets as in [4].

1.2 What Students Are Expected to Do

The project was not only designed as an approach to parallel programming, but also to emphasize the benefits of specification and validation prior to programming. In particular, the students were asked to produce a graphical model, having the same appearance as the physical railway. This is not required for aesthetic reasons but because it greatly helps to understand whether a configuration of the railway is correct or not. This eases a boring and error-prone task of synthesizing a long sequence of transitions. It represents an important benefit for debugging.

So far, hierarchical coloured Petri nets [9] have been chosen as a model, due to their tool support for hierarchies, simulation, and occurrence graphs (also called reachability graphs/trees and state spaces). Hierarchies allow a structured design, where the top-level net reflects the hardware layout. The use of high-level nets permits both capturing several cases by a single transition and representing the parameters of trains and track sections by one place. The use of an ordinary net leads to unreadable intricate models. The tool used to develop and analyze this model is Design/CPN [11].

Two views were considered in order to use the features of the physical railway. The first view, which is named *real railway view*, is meant to operate as a real railway, with the same rules. Effectively, the students are asked to design a system where each train is assigned a route. They must prove that some safety requirements are satisfied (no collision, no more than one train per section) as well as operational requirements (for example no deadlock). Once the model has been proved correct, the students have to figure out how to translate the net into a set of processes, synchronized by means of semaphores, which can control the model railway. The transformation should be systematic, such that its soundness and the preservation of properties can be justified.
The second view is not intended to mimic a real railway system. Rather, it is related to an adaptive routing system, and henceforth is named the *adaptive routing system view*. The behavior of trains adapts to local conditions. Namely, at each switch, the train's route can be chosen among several tracks and a train may even go back when it cannot continue forward. Although surprising at the first glance, such behavior of trains offers several complex routing possibilities and thus becomes a challenge. The students must design a routing policy so that the same safety and operational requirements as before are fulfilled. Then they have to derive from the net a program for a controller, and implement it on the railway system.

Whichever view is chosen, a validated piece of software controlling the railway system is expected, that must be both free from deadlocks and collisions. In this paper, we describe only the *adaptive routing system* view; Our reasons for restricting our description are discussed in Section 3.6, where we discuss the *real railway system view.*

1.3 Outline of the Paper

After a short presentation of the model railway in Section 2, we will detail in Section 3 the modelling and analysis process conducted by the students. Section 4 explains how a program controlling the hardware is derived from the net. Section 5 concludes the paper.

2 The Model Railway

The model railway is depicted in Figure 1. It consists of about 15 meters of tracks, divided into 16 sections (blocks B1 to B16) plus 2 sidetracks (ST1 and ST2), connected by four switches and one crossing. The way the trains can pass the switches and the crossing is indicated by the arrows in Figure 1. The traffic on all tracks can go both ways. Although one can notice that switch 1 (and also switch 2) is composed of two elementary ones, it is managed as a single unit, due to the short distance between the two physical components. 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 exit of a train. The orders sent to trains can be either stop or go forward/backwards at a given speed.

3 Specification Using Coloured Petri Nets

In this section, we describe the different steps followed by typical students (in fact, each project is conducted by a pair of students) for the second view, that is the adaptative routing system described at the end of Section 1. As a starting point, we give the students a net modelling a single train on a simple loop (three sections and no switch). This net only controls the possible moves of a train: wait for permission to enter a section, start and stop. The students are free to choose the initial position and direction of trains. They can also experiment with different initial conditions. Students are asked to proceed step by step in the construction of the coloured net representing the model railway. Each step consists of adding new features to the previous net, analyzing and correcting it. The resulting model is presented to a teacher during tutorials. The teacher then points out the pros and cons of the model, so that the net can be improved with respect to design and layout. The final net must represent the whole railway with a satisfactory behaviour for four or five trains. At the end of this section, we will indicate the differences with the real railway view (in Section 3.6).

The students' first model only describes the main loop, that is the railway without the 2 sidetracks and the crossing. The analysis of this net is performed for three, four and five trains. The initial marking in Figure 2 is a

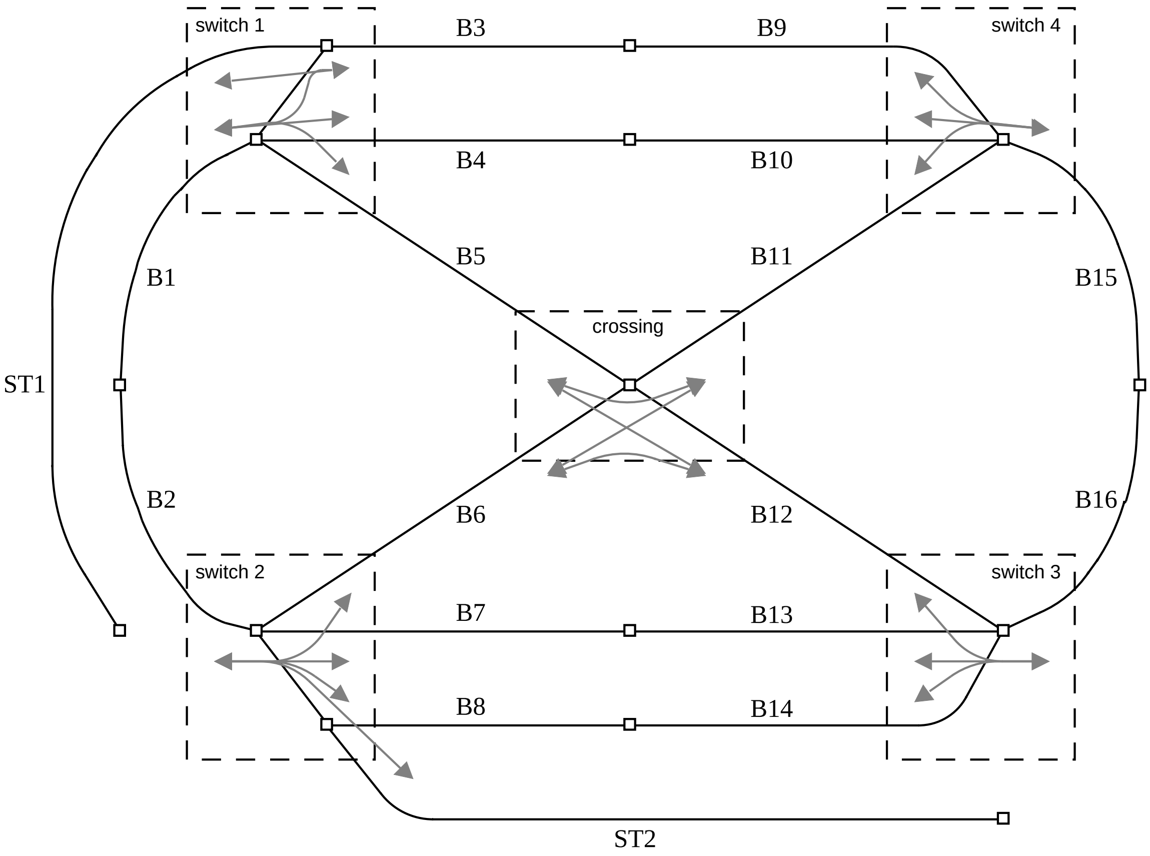

Fig. 1. The tracks of the model railway.

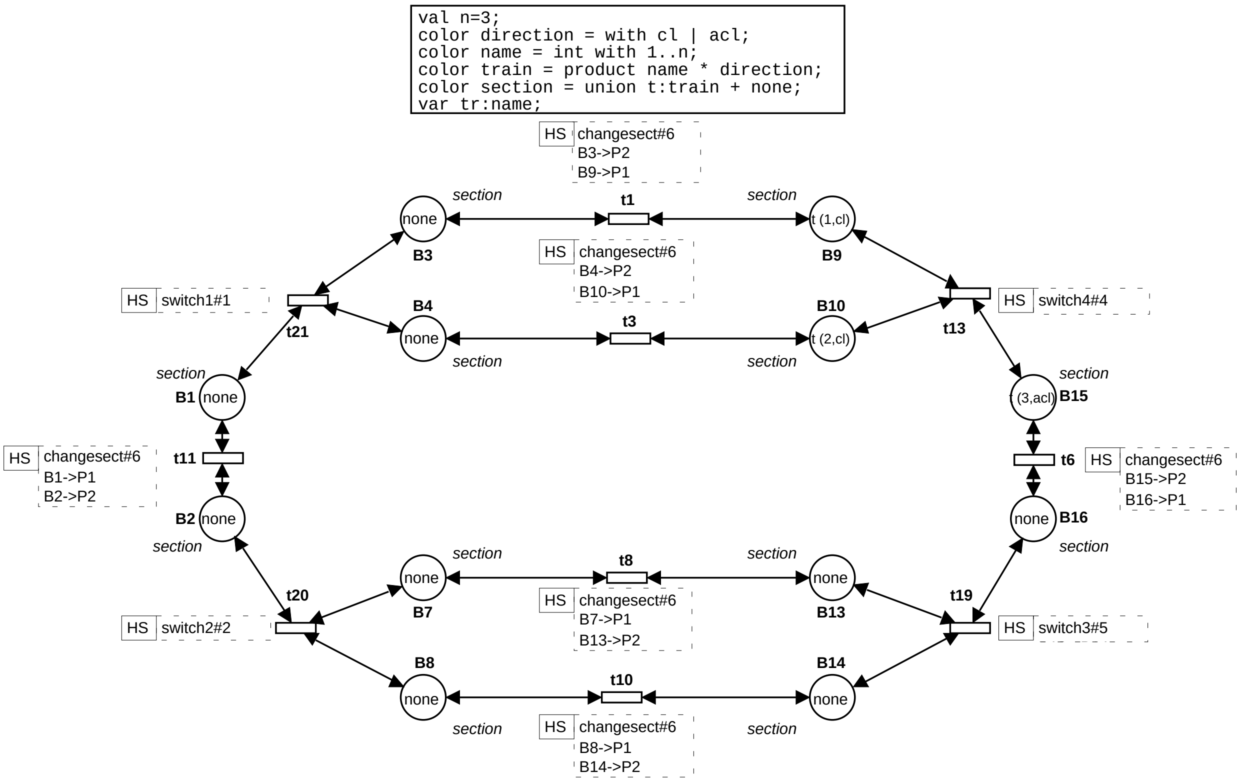

Fig. 2. The prime page of the main loop model.

deadlock situation as the three trains want to enter the same switch (modelled by transition t13). To avoid this, a train can change its direction when it is blocked (more precisely, when it is not possible for any train to pass the switch). Then the new model is analysed, uncovering eventually other errors, that are corrected. This process is repeated until we obtain a model which is satisfactory for five trains. All the corrections made correspond to a policy for managing the four switches and moving between adjacent sections. Then the two sidetracks are added and the analysis is done directly for five trains. Finally, the crossing is added. Generating the occurrence graph and checking properties of the complete model consumes both a lot of time and memory (for the model presented here, approximately 1.5 days and 800MB[1]). Hence the students are instructed to check the properties of the full model first with four trains and then with five trains.

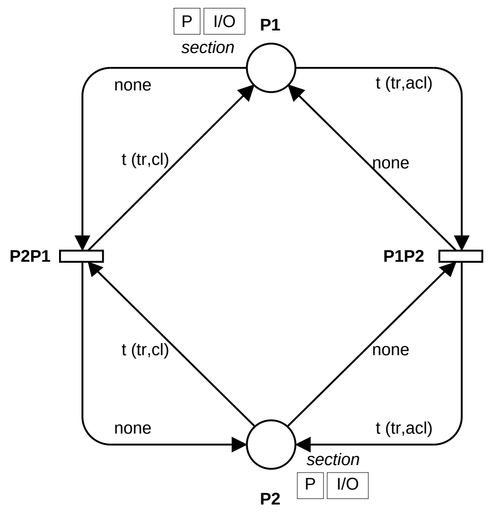

Fig. 3. Moving between two adjacent sections.

3.1 *Design of a Hierarchical Coloured Petri Net*

One of the requirements of the project is to obtain a model which can easily be understood. As we will see later on, this facilitates the understanding of errors encountered when analyzing the model using occurrence graphs. The corrections also become much more natural.

The use of page hierarchies was required: the prime page represents the whole railway, without any consideration of the policy used to move from one section to the next. This policy is described in subpages, corresponding to the different switches and moves between adjacent sections. A single look at this prime page shows the current state, i.e. where the different trains are located. The similarity between the physical railway model (Figure 1) and the prime page (Figure 2 for a partial representation and Figure 7 for the full one) is easily observed. The places represent the sections (they have the same names in both figures), while the transitions indicate the possible moves.

The nets presented in this paper are very similar to those obtained by the best students.

3.2 *Model of the Main Loop*

The students are asked to design their models step by step. Therefore, they start with just the main loop, that is the railway without the two sidetracks and the crossing in the center.

The prime page of the first net is shown in Figure 2. The colors (data types) of tokens within places are defined in the *global declaration node* (boxed text at the top of Figure 2). After declaring a constant which represents the maximal number of trains, four colors are defined. First, the `direction` of a train can be either clockwise (`cl`) or anticlockwise (`acl`). The `name` of a train is an integer between `1` and `n`. To follow the real system as much as possible, students generally choose to describe trains with both a name and the direction they are running, namely clockwise (`cl`) or anticlockwise (`acl`). So a `train` is a pair `(name,direction)`. Each place represents a railway section with the corresponding name. It always contains one token of color `section`, with a value characterizing the state of the section, that is either a train is in the section, or the section is empty. This is expressed with the union type:

```
color section = union t:train + none;
```

All the transitions are substitution transitions, meaning that their behaviour is described on the associated subpage. For example, transition t1 of Figure 2 has a box next to it with the name of the subpage to be substituted to this transition (`changesect`) and the correspondence between the names of input and output places of both nets (place B3 of Figure 2 corresponds to place P2 of Figure 3); the double arcs on Figure 2 indicate that the places are both input and output of the subpages. The two transitions in Figure 3 correspond to the two directions of the rail sections. For instance, transition `P1P2` may fire only if there is a train going anticlockwise (`t (tr,acl)`) in place `P1` and no train (`none`) in place `P2`. Firing this transition results in moving the train from place `P1` to place `P2`.

The substitution transitions (Figure 2) are of two kinds:

– those allowing movement between two adjacent sections with no choice; a reasonable hypothesis is that all transitions of this kind have the same behavior. Therefore, all these substitution transitions will refer

[1] all the results in this paper were obtained on the same machine: a Linux PC Pentium II 450 MHz with 256 MB of memory. The computation times are those given by Design/CPN.

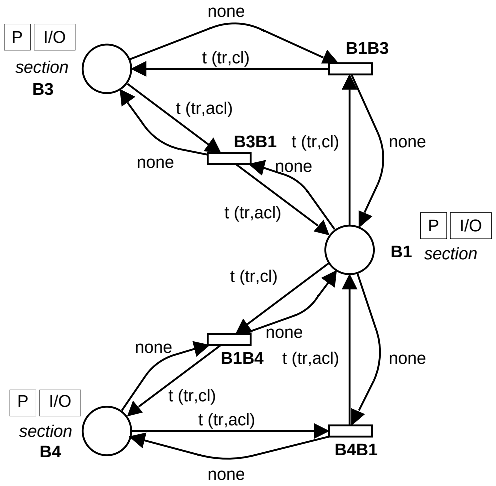

Fig. 4. A simple switch.

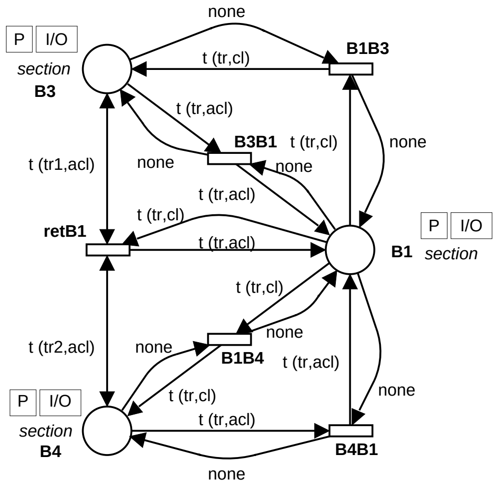

Fig. 5. New policy for switches.

to the same subpage. Initially, we will assume that a train can move to the next section if this section is empty, as depicted in Figure 3.

– the switches which can permit movement either from a unique section to two different ones or vice-versa. This is described in Figure 4, where the net corresponds to one possible direction of trains (switch 1). Another page describes the other directions of trains. These nets could have been merged into a single one, with a place containing the direction used by the switch. Such a choice would have lead to a less intuitive interpretation of the prime page, and to a unique page used by all the switches. Moreover, when enhancing the model to add the other parts of the railway, the switches become different. Similarly, some students have used two substitution transitions corresponding to the net in Figure 3 to model the simple switch, but then, when changing the policy for moving from one section to the next (as required after further analysis, see Section 3.3.2), the control of the switches needs to be modified. This is often undesirable. Once the net is completed, students start examining its behavior. The marking (with 3 trains, two going clockwise in places `B9`, `B10` and one anitclockwise in place `B15`) given in Figure 2 is a deadlock. Thus, a more complex policy for controlling the switches must be designed, as depicted in Figure 5. The switches are (for the moment) composed of two tracks arriving from the same direction (such as `B3` and `B4`) and one from the other side (as `B1`). When 3 trains arrive simultaneously to the switch in a deadlock situation, the new policy directs the train on the single track side (for instance the train on `B1`) go backwards. Such a possibility makes sense only in the adaptive routing system, but not in a real railway system. This is modeled by transition `retB1` in Figure 5.

The marking representing three trains will enable only the new transition, and thus only the train going anticlockwise will change its direction. Thereafter, all the trains will go in the same direction, that is clockwise. At this point, an extra train is added, and the new model is analyzed.

3.3 Verification with Four Trains

So, an improved policy for switches is used and a fourth train is added by changing the initial marking of place `B3` to `t(4,acl)`, and the value `n=4` in the declaration node.

The analysis of the model is performed by means of Design/CPN occurrence graph tool [10,11]. Once the graph is generated, a standard report and some additional properties are checked. This reveals nodes satisfying undesired properties, as explained later. To help understand the configuration of trains, the feature of Design/CPN which transforms a state of the occurrence graph into the corresponding marking in the simulator is used. Thus, the prime page is updated with the undesired marking, and the user can see the locations of trains.

3.3.1 Results of Analysis

After the occurrence graph generation, because of the huge number of states, students realise that the train names contribute to the state space explosion, although the names are useless from a verification point of view. Indeed, in the problem considered, it is not necessary to know where a particular train is, but only that there is a train in a particular section, and also where it is heading. This could be formalized using symmetries (for example permutation of names), but there is not enough time to teach the theoretical and practical aspects of this.

Moreover, the marking equivalence derived from such symmetry is quite obvious.

Occurrence Graph with Trains Names. The occurrence graph obtained for four named trains contains 21,574 nodes and 72,026 arcs. It is computed in 334 seconds. The strongly connected components (denoted by SCC are sets of mutually reachable states, see also [10]) graph has 1,243 SCCs and 6,666 arcs. It is calculated in 18 seconds.

Occurrence Graph without Train Names. When removing the train names (only the direction of the train is represented in the token value), the occurrence graph obtained, for four trains, has 2,166 nodes and 7,157 arcs. It takes 5 seconds to obtain it. The SCC graph has 237 nodes and 1,245 arcs. It is computed in 1 second.

The next step of the project is the verification of the net. The students must formalize correctness, and indicate the properties that should be satisfied. The issue is to select among the properties provided by the standard report of Design/CPN those, which can be used in order to prove the correctness of the model. The standard report functionality produces a textual file containing results for the standard net properties (such as bounds, dead markings, and liveness). It turns out that most of the desired properties are safety properties which can be checked using only the standard features.

The results and their interpretation are as follows:

- all the lower and upper best integer bounds are 1. Thus *there is always exactly one token in each place*;
- all the best upper multi-set bounds are `1‘t(cl)++ 1‘t(acl)++ 1‘none` (one train `cl`, one train `acl`, no train). This property, analyzed together with the previous one, shows that *each section can contain either exactly one train going one way or the other, or none*;
- there are *6 dead markings*. They represent situations where all trains are stuck. At first, this seems often surprising. The list of all dead markings can be obtained by evaluating function `ListDeadMarkings`. Then, each of these markings can be visualized on the prime page, using the feature of Design/CPN which puts a given marking of the occurrence graph into the simulator. Three of the dead markings have a train going clockwise in `B15` and a train going anticlockwise in `B16`. The three other ones are similar, with sections `B2` and `B1`.

The conclusion is that the policy for moving between two adjacent sections needs to be improved.

3.3.2 Policy Adopted

In order to avoid the situation described above, trains are required to go backwards, as modeled by transition `retboth` in Figure 6, with the following conditions:

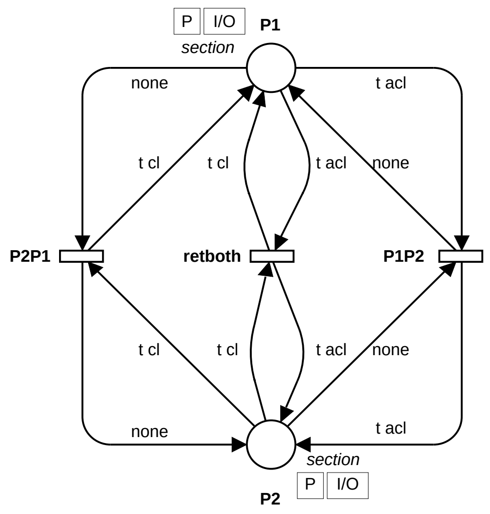

Fig. 6. New policy to move between two adjacent sections.

- if both trains return where they came from, they will not travel in the same direction, thus such a policy should limit or delay the possibilities for all trains to go in the same direction;
- both trains are treated in the same manner, there is no priority.

The new model is then analyzed for four and five trains. The following section discusses the results for five trains.

3.4 Verifying the Model with Five Trains

A fifth train going anticlockwise is added in place `B2`.

3.4.1 Results of Analysis

The occurrence graph obtained for five trains (without train names) has 24,556 nodes and 97,020 arcs. It is computed in 430 seconds. Its SCC graph has 615 nodes, 3,128 arcs and is calculated in 18 seconds. The properties obtained from the standard report, show that the bounds are the same as previously, *there is no dead marking nor home marking.*

This last property may be quite intriguing. Design/CPN built-in facilities make it possible to obtain more information about the terminal SCCs. The following functions (where the result is given after the arrow) were used to characterize the SCCs:

```
PredAllSccs SccTerminal;        -> [~615,~534]
length(SccToNodes(~615));       -> 792
length(SccToNodes(~534));       -> 792
```

`PredAllSccs SccTerminal` provides the list of all terminal SCCs. There are two such components, numbered ˜615, and ˜534. Then, the number of nodes in each of these terminal SCCs is 792. After looking at a marking of each component, the students conclude that this

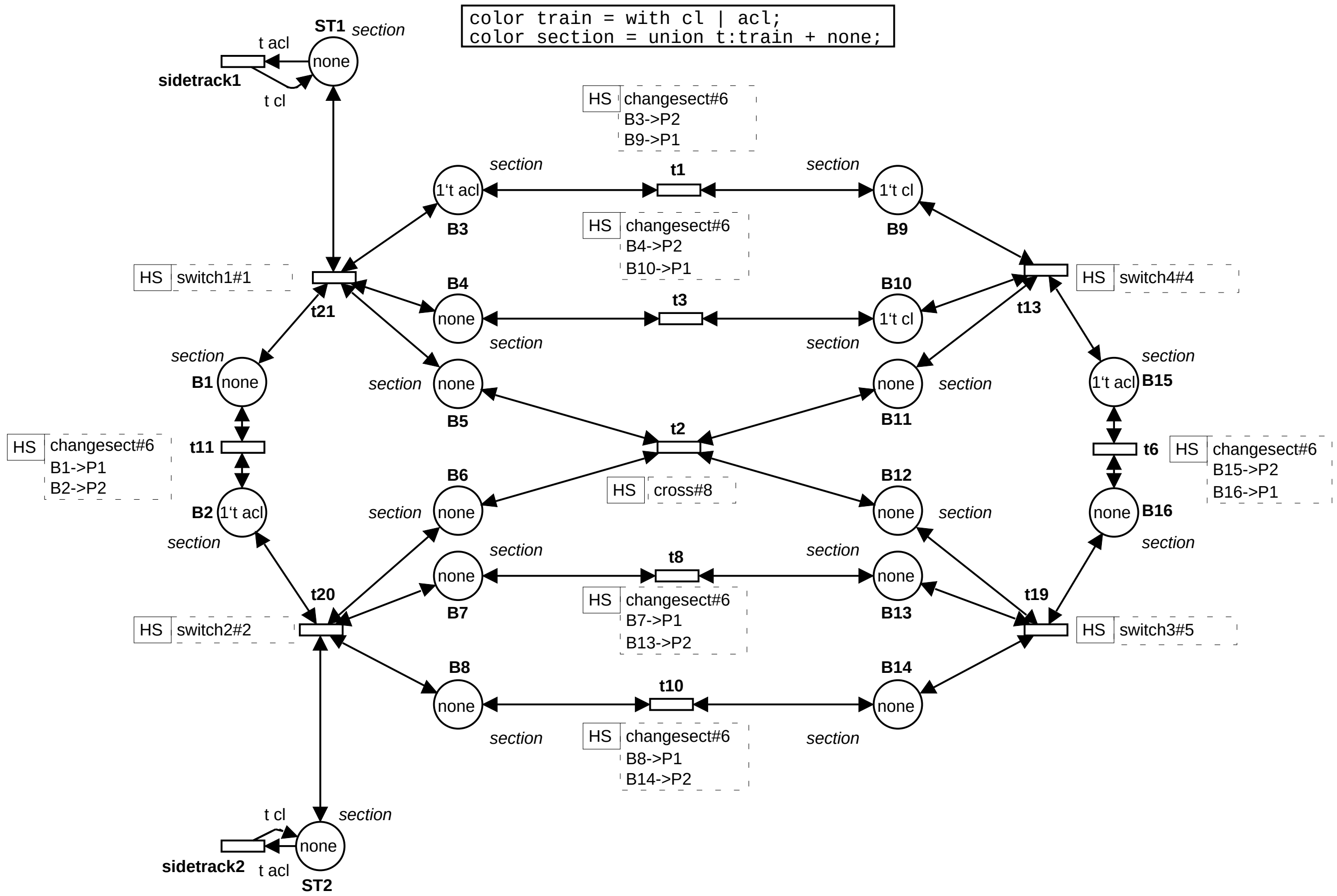

Fig. 7. The prime page of the model railway hierarchical coloured Petri net.

is a normal situation: in SCC ˜615, all trains go clockwise while in SCC ˜534, all trains go anticlockwise. The two SCCs remain separated as, considering the routing rules adopted, a train cannot go backwards if it does not meet a train going in the opposite direction. So, such situations are acceptable with respect to the initial requirements.

3.5 The Complete Railway

When the main loop is verified, the model can be enhanced by adding the two sidetracks and the crossing.

3.5.1 Final Model

The prime page of the model of the complete railway (see Figure 7) has 6 additional places corresponding to the 2 sidetracks and the 4 inner sections, and one extra substitution transition which models the crossing.

A subpage, shown in Figure 8, is added to represent the inner crossing in the railway. It takes into account all the possible moves on the inner crossing, as described in Section 2. Moreover, the case where four trains want to enter the crossing – and will then be blocked – is treated by forcing the train on `B5` to go backwards. Here we could have chosen a symmetric solution but we wanted to avoid changing directions of several trains (this might cause trains going back and forth). Choosing `B5` instead of `B6`, `B11` or `B12` is quite arbitrary.

The four switch subpages are modified to take into account the new possibilities, but their general policy remains the same, just taking into account more possibilities for a train to enter and exit a switch. The nets in Figures 9, 10, and 11 illustrate of the complexity of the complete model.

The fourth switch can easily be deduced from the third one, by changing the place names and directions of the trains.

3.5.2 Results of Analysis

As considerable time is needed to build the occurrence graph of the full model, the graph is first built with four trains and then with five.

The occurrence graph with four trains contains 48,957 nodes, 228,790 arcs and was calculated in 2,991 seconds (50 minutes). Its SCC graph has 1 node, no arc and is obtained in a bit less than 2 minutes.
The standard report gives us the same integer and multiset bounds as before. *All the reachable states are home*

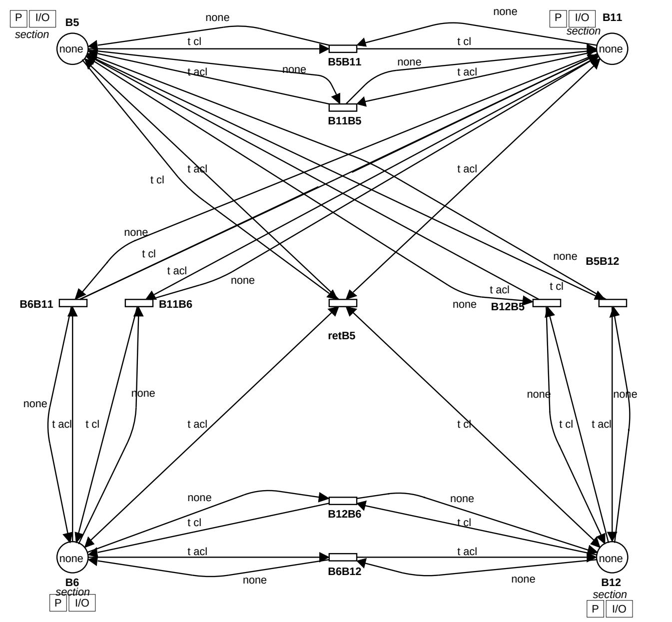

Fig. 8. The subpage modeling the inner railway crossing.

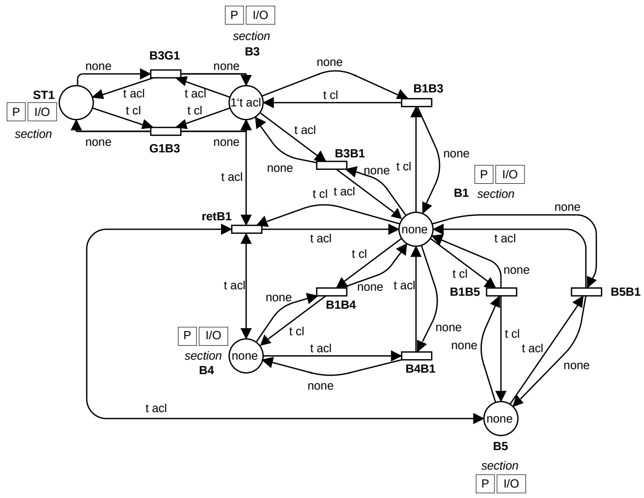

Fig. 9. The first switch (transition t21 of Figure 7).

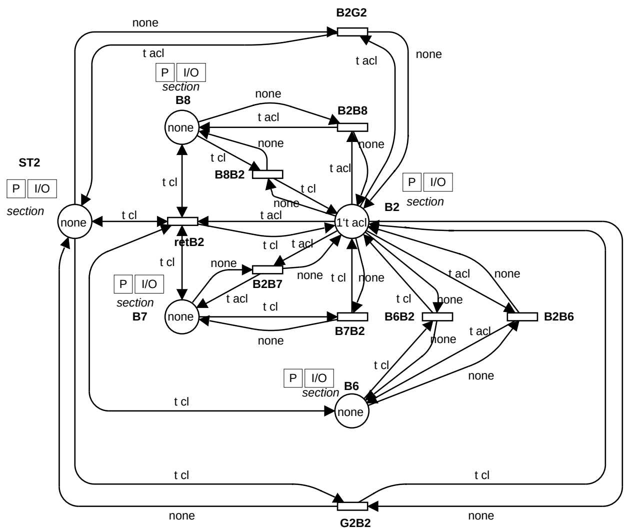

Fig. 10. The second switch (transition t20 of Figure 7).

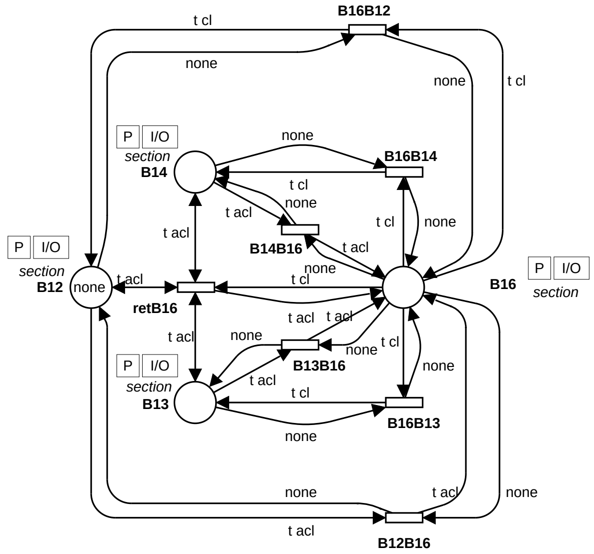

Fig. 11. The third switch (transition t19 of Figure 7).

markings. This is interesting, because it shows that any reachable train distribution can always be reached again. There is no deadlock. But *transition* `retB2` *of subpage* `switch2` *is dead*. This means that when four trains start as described by this initial marking, the situation where a train going anticlockwise on section `B2` is blocked can never occur. When looking at the design of the railway, it can be noticed that this is normal as there are 4 possibilities for this train to go forward, and only three can be occupied by the other trains. The question now is: what happens when a fifth train is added?

The occurrence graph with five trains has 274,082 nodes, 1,500,384 arcs and was computed in 103,221 seconds (1 day, 4 hours and 40 minutes, with intensive use of the 1GB swap space), approximately the same time was necessary for the SCC graph which finally contains 1 node and no arc. This informs the students that, as for four trains, all the reachable states are home markings. There is no deadlock and *all transitions are live*. Thus the model meets the initial requirements: allows up to 5 trains, avoids collisions and deadlocks.

3.6 *Modeling the* Real Railway System

Although the *real railway system* view is more realistic and may seem more appealing, tackling it raises several difficulties. When this view is considered, a route must be assigned to each train. This has three consequences: First, when a train enters a switch, it must exit it as specified in its own route, and not by a randomly chosen exit. As the second consequence, trains are characterized by their route plan and can be distinguished one from another. Finally, in order to avoid deadlocks (as required in the specification of the project, Section 1.2), a train may have to book several sections in advance: for instance, if a train in B7 wants to enter B2, it should also make the reservation for B1, otherwise it can be blocked by a train coming the other way around. As concerns modeling, a transition must not only be connected by the arcs to places representing the next and/or previous resource, but also to the places representing the resources necessary one or more steps later. Hence, the net is graphically less similar to the physical railway. Also, it is more difficult to create a hierarchical net as there are special transitions associated with each train.

The number of states in the occurrence graph strongly depends on the length and the complexity of the routes designed by the students. Another big difference with the second approach is that verification mainly consists of deadlock detection between trains. It is obvious that complex routes and the simultaneous reservations of four sections can lead to deadlocks that are very difficult to predict. To conclude, the complexity of the model and the possibility to fully check it depends a lot on the options chosen by the students. In practice, we are led to impose severe restrictions, lessening the credibility of the verification process. Moreover, the model is less readable, which also decreases the educational aspect. This is why, after experimenting with both views, we chose to describe the other view (*adaptive routing system*).

4 From the Specification to the Implementation

When the net model of the railway has been analyzed, and its desired properties proved, the model needs to be translated into a program controlling the railway model, which follows the net behavior as closely as possible. To do so, there are two solutions: the first one consists of writing a controller (a Petri net simulator), the second is to decompose the net into a set of synchronized processes. Students are naturally inclined to use the first solution, which is closer to sequential programming and automata theory, but, after a deeper insight, some of the students attempt to try the second solution. We shall now describe both solutions in more detail.

In the simulator approach, the color sets and associated operators are translated into `C` data structures and functions. Then, each transition is split into a boolean precondition function and a postcondition function. The former permits checking if the transition is firable, while the latter is called when firing occurs to change the marking and send orders to the hardware. The core of the simulator is an endless loop which, at each step, scans the transition list to see which transitions are firable, and then fires some of them. It should be noticed that, contrary to what happens in a general simulator, the binding of variables of transitions is quite easy since each place contains exactly one token. Nevertheless, when several transitions are enabled, a fair choice must be provided, using, for instance, a random number generator. A separate process is in charge of reading the information from the hardware sensors, and updating the state variables accordingly.

The second approach is more natural when using the *real railway view* because the trains are seen as independent entities which travel along a route, but can also be applied to the *adaptive routing system view*. Although this approach is less intuitive than the previous one, it permits a concurrent programming style, which is one the goals of this course. Each token `(i,direction_i)` of color `train` in a place `P` can be considered as a state of a process train `i` (train `i` is in section `P`). This process may evolve from one state to a next one (corresponding to a place `P'`) by executing a procedure which implements the firing of the transition between places `P` and `P'`. So the whole token game is simply distributed among the train processes. The atomicity of a procedure (transition) is sufficient to have exactly the same behavior as the net. The atomicity can be guaranteed by enforcing mutual exclusion, which may be implemented using for instance Dijkstra's semaphores [5]. Mutual exclusion should not be applied to check the contents of each place successively, but on the whole transition. If a process is blocked, it leaves the critical section and becomes suspended on a semaphore until it is activated by another process. With this approach, no fairness problem appears at first, provided that semaphores have FIFO queues. However such a problem arises when a train arrives at a fork with several possible exits. So a mechanism to choose randomly which transition to try must be added. As in the previous approach, a separate process handles the hardware inputs.

The program obtained is then downloaded on the PC which monitors the train model and the students may conduct some experiments. The final program is often quickly obtained, after some tuning (speed of trains), addition of hardware commands (switches, crossing), and detailed improvements (traffic lights). The students can see that no real problem arises, except from faulty hardware behaviour (if a sensor does not *see* a train, the train is *out of control* and must be physically removed as soon as possible). However, most of the time a long run results in a periodic behavior, even if the theoretical analysis predicted a less rigid scheme. This is due to the fact that even with a fair program, trains arriving first at critical points are also served first, and the delay to run through a block depends mainly on its length. So, the model train can adopt only a subset of behaviors of its theoretical model. The exact analysis would require temporal aspects, but this has not been attempted yet.

5 Conclusion

In this paper, we have discussed a teaching experience based on a railway model and high-level nets.

In spite of its toy aspect, the train model forces students to study and deal with all the effects which make parallel programs error prone: critical sections, deadlocks, fairness, time and combinatorial explosion. Moreover, there is very strong evidence that such programs cannot be developed from scratch (bottom-up style) as students tend to do, but must be rigorously modeled and analyzed before implementation. An additional benefit from this project is that students are exposed to a process control problem.

This project has both positive and negative aspects. Coloured Petri nets and DESIGN/CPN proved to be easily handled by students. They could even use advanced features such as hierarchies and simulation with profit to perform first level debugging. Students usually think that simulation is enough to guarantee the correctness of their model. Then we tell them to try out the occurrence graph tool ; they are generally surprised to discover unexpected behaviour of the net, for example a deadlock. The possibility to transfer a marking from the occurrence graph to the net in the simulator is useful to visualize a particular situation. Having always exactly a single token per place is quite natural for students which are used to programming with variables and procedures. However such a modelling style makes the use of invariant analysis less obvious. This is not a real drawback, because, from our other experiences, finding and interpreting place invariants is possible only for more experienced and trained students (the lack of tool does not help either). Another difficulty is due to the state space explosion problem. Student design models which often overtake the tool limits, forcing us to supervise the different steps of the project in order to help the student find the cause of the state space explosion (unnecessary variables, too complex tokens colors, . . .).

In the students opinion, it appears that handling a real system helps and motivates both modeling the system and formalizing the properties that it should satisfy. However, the students are disappointed by the state explosion problem which becomes an issue quite early during the verification. Thus only restricted or more abstracted problems can be validated.

A further step would be to manage temporal aspects. An automatic code generation feature could have helped a lot for the last step of the project, that is the implementation part[2]. However, students would not have been as aware of difficulties encountered when going from the specification to the implementation.

[2] DESIGN/CPN provides SML code generation, but it would be more difficult to interface it with the pieces of code controling the hardware.

Acknowledgements. Thanks to all the people who helped us carrying out this project and improving this paper. In particular Kurt Jensen, the anonymous referees, and all the people who discussed the educational issues after the presentation at the workshop on high-level Petri nets, and last but not least, the students who worked hard on the project.

References

1. J. Billington and C. Janczura. Generalizing results for any legal initial configuration of a railway network. *Electronic Letters, IEEE*, 32(24), 1996.
2. J. Billington and C. Janczura. State space analysis of a railway network. In *Proc. IEEE Int. Conf. Systems, Man, and Cybernetics (SMC'96), Beijing, China, October 1996*, volume 3, pages 2386–2391, October 1996.
3. S. Christensen and K. H. Mortensen. Teaching coloured Petri nets - a gentle introduction to formal methods in a distributed systems course. In *Proc. 18th Int. Conf. Application and Theory of Petri Nets (ICATPN'97), Toulouse, France, June 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 387–406. Springer, 1997.
4. W. M. P. Van der Aalst and M. A. Odijk. Analysis of railway stations by means of interval coloured Petri nets. *Real-time Systems*, 9(3):241–263, 1995.
5. E. W. Dijkstra. Solution of a problem in concurrent programming control. *Communications of the ACM*, 8(9):569, September 1965.
6. The Esterel language. `http://www-sop.inria.fr/meije/esterel/esterel-eng.html`.
7. W. Hielscher, L. Urbszat, C. Reinke, and W. Kluge. On modelling train traffic control in a model train system. In *Proc. 1st CPN Workshop*, DAIMI PB 532, pages 83–101. Aarhus University, 1998.
8. L. Jansen, M. Meyer zu Hörste, and E. Schnieder. Technical issues in modelling the European train control system. In *Proc. 1st CPN Workshop*, DAIMI PB 532, pages 103–115. Aarhus University, 1998.
9. K. Jensen. *Coloured Petri Nets: Basic concepts, analysis methods and pratical use. Volume 1: basic concepts.* Monographs in Theoretical Computer Science. Springer, 1992.
10. K. Jensen. *Coloured Petri Nets: Basic concepts, analysis methods and pratical use. Volume 2: analysis methods.* Monographs in Theoretical Computer Science. Springer, 1994.
11. META Software and Aarhus University. *Design/CPN 3.0*, 1996. Also available as: `http://www.daimi.au.dk/designCPN`.
12. Spin - formal verification. `http://netlib.bell-labs.com/netlib/spin/whatispin.html`.
13. Information systems train laboratory. `http://wwwis.win.tue.nl/tallis/`.
14. M. Meyer zu Hörste. Modelling and simulation of train control systems using Petri nets. In *Proc. 3rd FMERail Workshop*, 1999.