Proc. Estonian Acad. Sci. Phys. Math., 2003, **52**, 4, 378–393

Extracting high-level information from Petri nets: a railroad case

Thor Kristoffersen[a], Anders Moen[b], and Hallstein Asheim Hansen[c]

[a] Norwegian Computing Center, P.O. Box 114 Blindern, N-0314 Oslo, Norway; thor@nr.no
[b] Department of Informatics, University of Oslo, P.O. Box 1080 Blindern, N-0316 Oslo, Norway; andersmo@ifi.uio.no
[c] Department of Computer Science, Buskerud University College, P.O. Box 251, N-3603 Kongsberg, Norway; Hallstein.Asheim.Hansen@hibu.no

Received 10 February 2003, in revised form 21 May 2003

Abstract. We show how useful simulation and control applications can be built on Petri nets by adding an object-oriented transformation framework called “views”. Petri nets were integrated into the Common Lisp Object System in such a way that the usual object-oriented features, such as classes, object creation, encapsulation, and inheritance, also work for Petri nets. This modified object system allows views to be implemented as collections of methods on the modified classes. Views are used to extract and present relevant domain-specific aspects of a Petri net model. This functionality was implemented in our prototype Petri net engine called Andromeda. As a test case we modelled the Oslo subway as a hierarchy of railroad specific components, and built a composite view showing train and passenger movements.

Key words: coloured Petri nets, simulation, object-orientation, formal modelling.

1. INTRODUCTION

Coloured Petri nets (CPNs) have been demonstrated useful in simulating, visualizing, modelling, executing, monitoring, and analysing concurrent systems, including workflow and logistic processes. In this paper we suggest an integration architecture that can make CPNs suitable for use in large and complex real life applications. We show how to construct domain specific components and observe their behaviour using CPNs[1]. In particular, railroad systems are decomposed into their constituents and investigated in a precise way using CPNs.

[1] An earlier discussion of the modelling of railroad systems using a CPN-based approach can be found in [1].

The paper presents work in progress, with special focus on two problems in dealing with large Petri net systems: observation and construction. Manageable ways to observe the structure and behaviour of large-scale simulations in a systematic way are called *views*. To support the construction of large-scale systems, our approach is object-oriented, providing tools to define domain-specific classes, and support for inheritance, encapsulation, and interfaces.

2. PETRI NETS

Petri nets were introduced by Carl Adam Petri in 1962 [2]. Many variations of Petri nets exist, but we have based our work on CPNs [3] with slight modifications.

There are many calculi and languages that model concurrent behaviour, both symbolic and graphical[2]. Petri nets provide perhaps the simplest graphical formalism, and are also one of formal frameworks that have been investigated most extensively.

Petri nets are a well understood theory for modelling concurrent systems. It has a graphical notation which makes it possible to build Petri nets with graphical tools. Petri nets are Turing complete, so they are exactly as powerful as programs but much easier to analyse, since both states and transitions between states are explicitly modelled. General introductions to Petri nets are given in [4,5].

In this section we give an informal introduction to some of the key concepts in Petri nets. For a full and formal description of CPNs, see [3,6,7].

A Petri net is a directed graph in which each node is either a *place* or a *transition*, and each edge is an *arc* that connects one place to one transition. A place represents a state and may contain a collection of *tokens*, each being an object representing a process in that state. A token may be any data object, and possibly a composite one. A transition represents changes in process state. A particular distribution of tokens in the places of a Petri net is called a *marking*. A transition has *input* and *output* arcs from input and output places, respectively. When certain conditions (to be explained below) are met, the transition is *enabled*. If the transition is enabled, it may *occur*. When a transition occurs, tokens may be removed from the places on its input arcs, and other tokens may be added to places on its output arcs.

Each arc has an *arc expression*. The set of arc expressions on the input arcs of a transition govern whether that transition is enabled or not. If it is enabled, those arc expressions may bind a set of variables to objects consisting of tokens in the input places, or parts of these tokens. When a transition occurs, the arc expression on an input arc determines which tokens are removed from the input place, and the arc expression on an output arc determines which tokens are added to the output place.

[2] Examples of symbolic calculi are Pi-calculus, ambients, CSP, and communicating automata. Among the best-known graphical languages are SDL, UML, MSC, and flow charts.

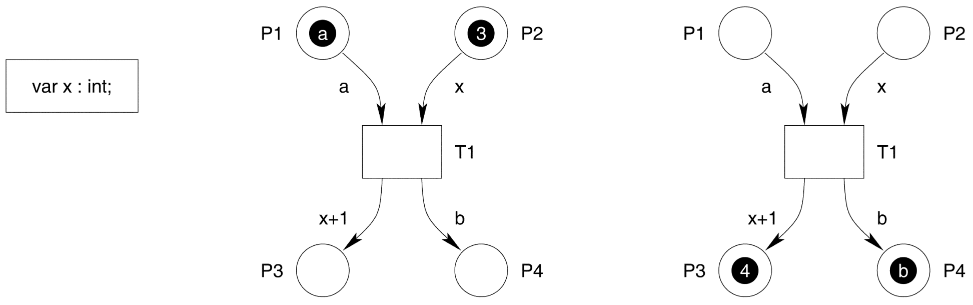


Fig. 1. Occurrence of a transition.

A transition may have a *guard* function which evaluates to a Boolean value, and the transition is enabled only if this value is true. The arc expressions and the guard may contain variables that are bound as described above.

The net in Fig. 1 consists of four places, $P1$, $P2$, $P3$, and $P4$, and one transition, $T1$. In this net the input places of $T1$ are $P1$ and $P2$, containing one token each. In the arc expressions, x is a variable, and the other symbols are constants. For $T1$ to be enabled, $P1$ must contain an a token, and $P2$ must contain an integer, x. Since this is the case, the transition may occur. When it occurs, x is bound to 3, and both the tokens are removed. Then output tokens are placed into the output places $P3$ and $P4$ as specified by the output arc expressions. The expression $x + 1$ evaluates to 4, which is added to $P3$, and the constant b is added to $P4$.

Petri nets may be extended with time in order to model processes with actions that take a certain amount of time on a given timescale. In timed Petri nets, every token has a timestamp, and a global quantity called the *model time* is introduced. A token may be consumed by an occurring transition if and only if its timestamp is less than or equal to the model time, in which case it is said to be *ready*. When a transition occurs, the timestamp of a token put into an output place may be incremented by a specified amount, thus making it unavailable for that amount of model time. The total effect of this mechanism is that in the context of the occurrence of one transition there is a delay between removal of a token from an input place and the disposal of a token into an output place. Since this delay is a part of an arc expression, it may be specified so that it depends on the values of the bound variables. It may also be given a random component, resulting in a stochastic delay.

3. THE ANDROMEDA SYSTEM

The main idea behind our work was to take a CPN framework and extend it with object-orientation and a type of simplifying transformations called “views”, so that useful applications can be built on top of it. In order to achieve full integration of CPNs and views, we built our own CPN framework from the ground up in Common Lisp. This framework provides all the functionality of CPNs and adds object-orientation. We first explain object-orientation and views, and then we describe the Andromeda architecture.

3.1. Object-orientation

We have called the Andromeda system *object-oriented* because it has three features that have been associated with object-oriented programming languages. First, *net classes* can be defined, and then at a later point instances can be created from them. Second, a net class provides *encapsulation* of a net and supports definition of an interface of externally visible places. Third, *inheritance* is supported in the sense that it is possible to define subclasses of an existing class by adding parts to the collection of parts defined by its superclasses.

In Andromeda, the Common Lisp object system was extended to allow definition of net classes. A net class is like a standard class in every way, but it has the additional property that it contains a *template* for a Petri net, i.e., a textual specification of its constituent parts (places, transitions, and arcs). This specification is sufficient that an executable Petri net can be created from it. When an instance of a net class is created, an associated Petri net is also created by creating parts in the engine according to the template of that net class. The Petri net semantics of the resulting Petri net is independent of the semantics of the object system, though.

By default, all the Petri net parts in an instance of a net class are externally inaccessible, but this can be overridden in the template by indicating specific places (and only places) as *external*, meaning that they will be externally accessible in an instance. This feature provides encapsulation and definition of external interfaces.

The usual class inheritance protocol was extended so that inheritance works not only for classes, but also for their templates. Given a particular net class, new parts can be added to its Petri net template by creating a subclass that defines those additional parts in its template.

In Andromeda, hierarchical Petri nets are supported through the use of *subnets* in templates. A subnet is an instance of a net class with a subset of its places externally accessible, so that it can be used in a template as an opaque module with a collection of associated places. The creation of a net class instance is recursive: if the template contains a subnet, an instance of the class of that subnet will be created. Thus, the result of a recursive instance creation is a hierarchical Petri net.

The tight integration of Petri nets with the object system was important in order to support views, which we will discuss next.

3.2. Views

In Andromeda, the modelled system contains information that can be divided into two categories. First, there is the object model, just as in any object-oriented program. This includes information stored in the slots of the objects, as well as information about the hierarchical relationships between the objects. Second, for each object there is an associated Petri net containing tokens. Thus, if we have information that is specific to a particular subnet, but not directly relevant to its Petri net semantics, we put that information into the slots of that subnet. A *view* is, in the abstract sense, a function that maps both of these types of information into some simpler data structure that captures a particular aspect of the modelled system.

A graphical representation of a view is called a *visual view*. Visual views can make a Petri net system user-friendly. Even though CPNs provide abstraction mechanisms over low-level Petri nets, this does not guarantee that it will be easy to understand the operation of the system as a whole. Since a Petri net model of a system is precise, it is possible to extract, as views to the observer, any kind of information about the behaviour that is represented in the model.

As a test case for our Andromeda system, we implemented a simulation of the Oslo subway, with stations, lines, trains, and passengers (further described in Section 4). In this application, the top-level Petri net models the entire railroad network, and within this net each station and line segment is modelled by one subnet, and within each station subnet, each platform is modelled by one subnet.

We define three net classes: `station`, which models a railroad station, `segment`, which models a stretch of tracks connecting a pair of stations, and `platform`, which models a platform. The top-level net consists of a collection of instances of `station` and `segment`.

The slots in the `station` class include information irrelevant to Petri net semantics, such as the name of the station and its coordinates on the map. When a specific `station` subnet is created, the name and coordinates are supplied in order to initialize its slots. The Petri net in the `station` class contains instances of the `platform` class.

The slots in the `segment` class include the length of the segment and the time used by a train driving through it.

Given the above railroad model, the following are typical applications for views:

– *Extract information about a given subnet at that hierarchical level.* For example, a view that extracts station relevant data for each station and line segment subnet, including the name of a station, the names of the lines it belongs to, the position of the station on the map, the length of a line segment, and whether a train token is currently in the station or in a segment. The graphical representation of this view would be a high-level map with names and some kind of marker to represent the presence of a train.

– *Flatten several hierarchical levels of subnets.* For example, a view that flattens two levels of the hierarchy in order to show a more detailed version of the above-mentioned map with individual platforms.

– *Compute a scalar value for a given subnet.* For example, a view that returns the total number of passenger tokens within a line segment subnet. The graphical representation of this value could be a bar graph on each line segment in the map, or a colour coding of the segments, showing the passenger density in that segment at any given point in time.

In practice, a view is made up of a collection of methods on the classes of the constituent objects. This kind of organization of views is facilitated by the fact that Petri net classes are just like standard classes. When a method that implements a view is called on an object, it may read the values of the slots of that object, just as in any object-oriented system. In addition, it may inspect any places that are directly contained in its net, and it may call methods on any of its subnets.

For instance, suppose that the presence of a train at a platform is indicated by a train token in any one of three different places. Then we write a `contains-train` method on the `platform` class, such that it returns true when one of those three places contains a train token. Now, if we want a `contains-train` method on the `station` class, it is written so that it returns true when the `contains-train` method returns true for any of its two contained platform objects. Finally, at the highest abstraction level, a visual map view displays a marker on a station when the `contains-train` method returns true for that `station` subnet.

3.3. Architecture

The Andromeda core architecture consists of three parts: class system, engine, and communication server.

The class system takes care of everything concerned with the definition of classes and computation of templates sufficient to create instances of classes. For example, when an instance of a class is created, it is necessary to compute an *effective template* based on all the templates in the superclasses of that class. The class system also handles the implementation of views since views are made up of methods that are specialized on net classes.

The engine executes Petri nets according to the standard rules of timed CPNs, except that dynamic typing is used instead of static typing. The interface to the class system is through an API that supports creation of Petri net parts (places, transitions, and arcs). Thus, when the class system is about to create an instance of a class, it reads the effective template and creates Petri net parts in the engine. If the effective template contains subnets, this process is recursive, as described in Section 3.

The communication server interacts with clients via the Andromeda application protocol over TCP/IP. Through the Andromeda protocol it is possible for external agents to connect to an executing subnet and observe its tokens, add tokens to it, and to remove tokens from it. A Petri net that is connected to the external environment in this way is called an *open* Petri net. The common application protocol makes it possible to create so-called *adapters* in order to interface Andromeda with other systems and technologies, like web and mail.

Andromeda also includes support tools, such as the Net Editor, in which it is possible to edit net classes graphically, and the Engine Visualizer, which supports visual inspection of executing Petri nets.

4. THE RAILROAD CASE

As a sufficiently complex test case for the Andromeda system, we chose to model the Oslo subway system. The Oslo subway consists of 101 stations and 5 lines, where trains run at 15 min intervals. In particular, we wanted to investigate two mechanisms of abstraction, *object-orientation* and *views*.

As part of our work with the Oslo subway application, we have designed and tested a number of selected railroad elements: segments, switches, and signalling systems.

4.1. Railroad components

In a realistic model of train systems, both planned and unplanned delays must be simulated. In the model, delays are specified in the railroad segments between stations, and also how much time the train spends in each station.

A railroad net is composed of railroad segments, switches, end segments, and platforms. A railroad system contains a railroad net, trains moving in the net, passengers travelling in the trains, signalling system, and power supply. The basic entity, the train, is represented as a composite coloured token, with four attributes: *train number*, *main direction* deciding the main direction the train moves, *local direction* referring to the current state of the train, moving either forward, backward or in stop state, and finally a set of *passengers* inside the train.

The railroad components consist of *segment places*, places representing physical positions and *move transitions*, transitions for moving a train from one segment place to the next. Figure 2 shows two basic railroad segments connected, consisting of three segment states s_1, s_2, and s_3, representing three physical points on the railroad, and *move* transitions t_1 and t_3 for moving the train forward, and t_2 and t_4 for moving the train backward. Since trains are vectors, with a main direction, we use $+$ and $-$ to denote whether the train is moving forward or backward.

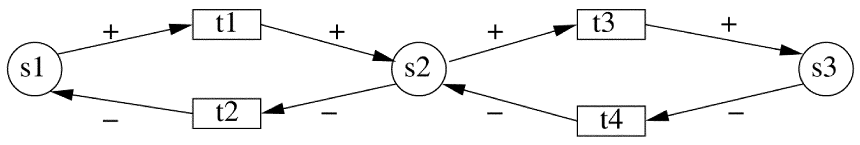


Fig. 2. Railroad segments.

Switch components consist of three segment places, *join*, *left*, and *right*, semaphore places L and R, controlling the routing of the tokens over the switch, as depicted in Fig. 3. The switch component must have an initial position, either left or right as indicated by the mutex pair of states L and R, either L or R carries a semaphore token initially. To change the switch position, one token is added by the user to the *Change* place.

We say that a *railroad line* is a connected graph of vertices and edges such that the vertices contain end line nodes, line segments, switches, and platform segments that connect the behaviour of the railroad line to the behaviour at the station and railroad segments. The edges contain segment transitions for moving trains. A *basic railroad net* is a set of railroad lines. In Fig. 4 we show a simple railroad circuit as a graph.

The switch places that permit input from the user are denoted by open circles. This graph can be directly translated to a corresponding Petri net, where each node in the graph is translated to a segment place, each arc is mapped to a pair of move transitions, and the switches are inserted directly, as can be seen in Fig. 5.

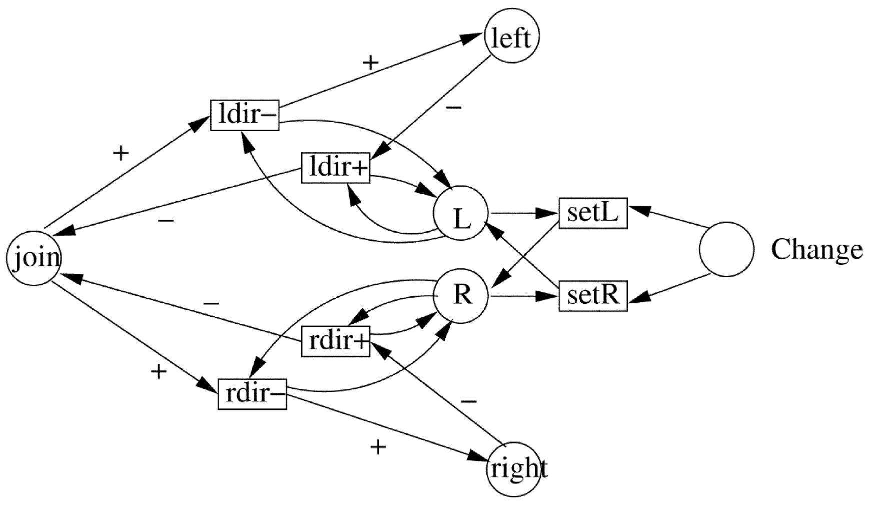


Fig. 3. The switch component.

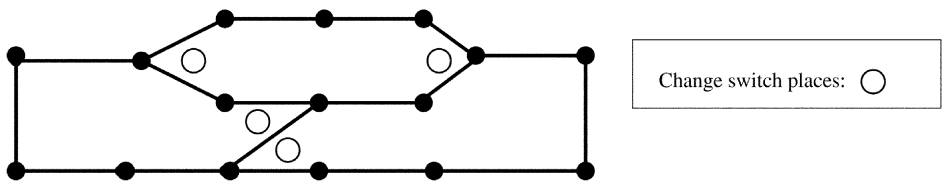


Fig. 4. A railroad graph.

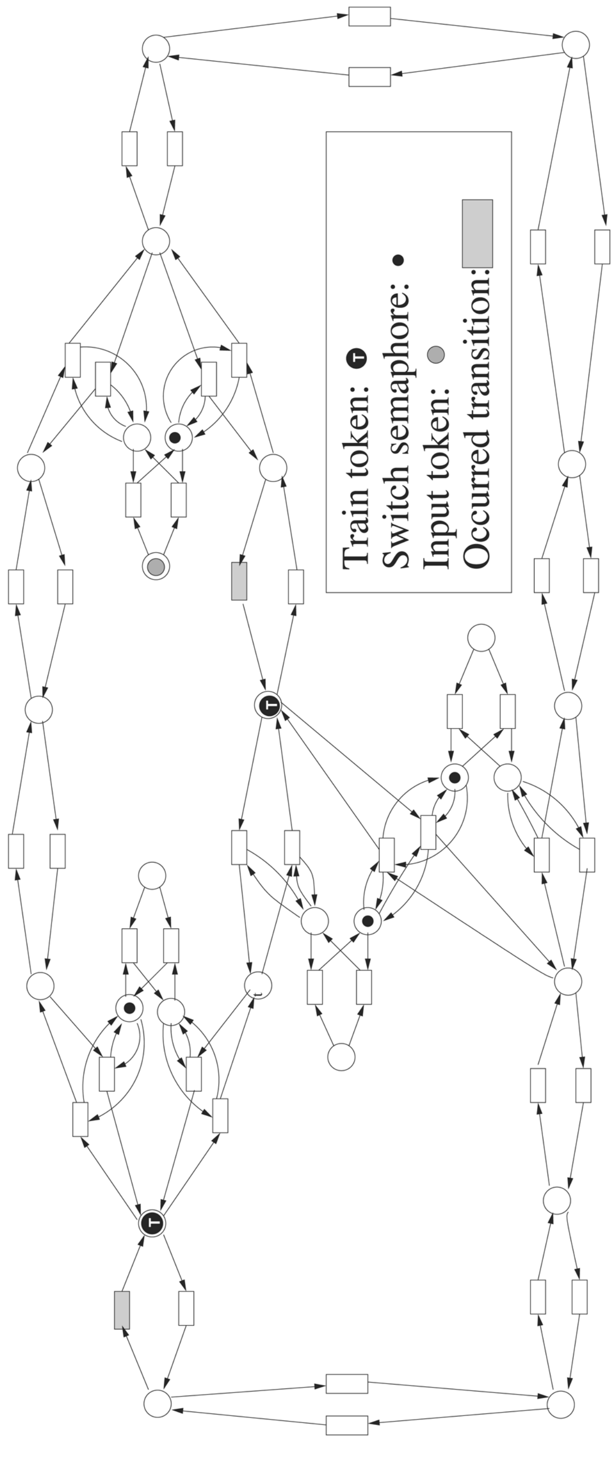


Fig. 5. Flattened railroad Petri net.

The railroad graph is depicted in Fig. 4 as a Petri net, with two trains moving in opposite direction. The rightmost switch is changed by the user, causing a change of the switch position in the next state.

4.2. Signalling system

A signalling system is modelled over the railroad net, extracting information from the segment states. In real railway systems there are several kinds of signals, dependent on the role the train is in at a particular moment. The signals at the station area differ from the signals used on a single railroad track to prevent train collision. There are also significant differences in the signalling system of the railways in Russia, Germany, and the United States.

A simplified HV system signal light, used to implement block sections in German railways, is modelled as two mutex nets, with upper signal lights *green* and *red*, meaning respectively drive and stop, and lower signal lights *yellow* and *green*. The lower signal lights have three states: *green*, *yellow*, and *no light*, indicating ready to drive, warning drive careful, and full stop (see Fig. 6).

To give a full presentation of actual signalling systems and how they can be represented using Petri nets is outside the scope of this paper. But as Fig. 6 indicates, signal components are easily augmented to the railroad graph through the five Push-connector places, for acting with the signal, and the state of the lights: green, red, yellow, and no light.

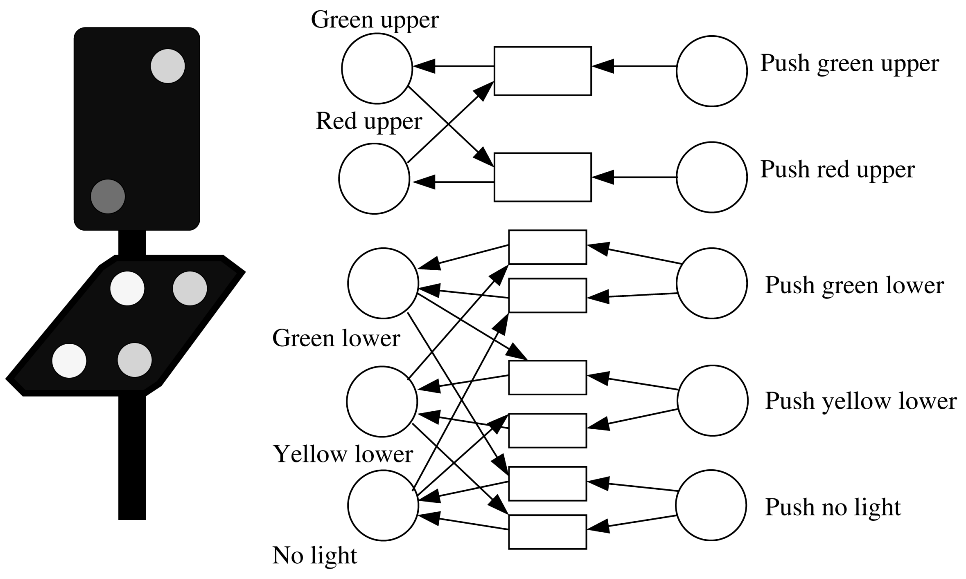


Fig. 6. Signal light.

4.3. Safety of the railroad

Note that in our simplified nets neither the line segment nor the switch represented in Figs. 2 and 3 provide *collision detection*, since trains might pass each other on the same physical line without noticing that two trains have crashed. Techniques to discover possible collisions and accidents, locally in the net, can be obtained in several ways by introducing either semaphore places, critical regions the trains run into before they enter the logical position in a line segment, or by other combined techniques.

A fundamental concept in railroad technology is the concept of block section (see e.g. [8]). An example of a fully automated signal light implementing block sections on the railroad is demonstrated in Fig. 7. As the train enters the block section (in Fig. 7 the block section starts at the place s_1) the light changes from green to red on the upper signal, and from green to no light on the lower signal. When the train leaves the block section, the light changes back to green on the upper signal, and from no light to yellow on the lower signal. Since the train is now in state s_2, this configuration indicates a warning to the next train to drive carefully, since the distance between the trains is so small that a collision can occur. The final state is when the train has left the block section, corresponding to state s_3. The signal now indicates to the next train that the block section is cleared, and it might drive at full speed.

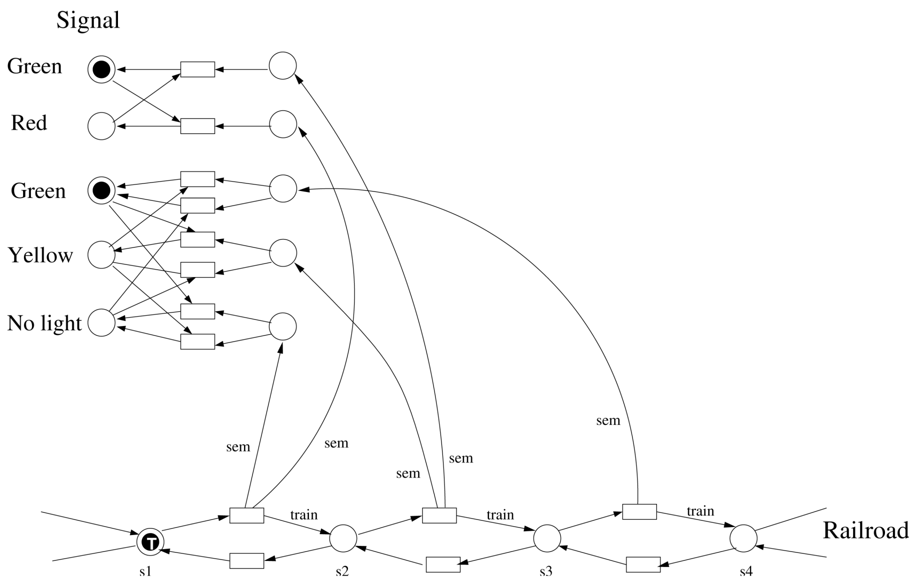


Fig. 7. Block section with signal light.

4.4. Views on railroads

One problem of simulating and monitoring railroad systems is to understand their behaviour. This is a good example to illustrate the usefulness of views. Relevant information should be provided so that supervisors could discover where the trains are, trains that are too late, alarms for detection of malfunctioning equipment and possible collisions, etc.

In the screenshot taken from our Oslo subway application (see Fig. 8) we present two distinct views, one showing the subway map and the other displaying a list of trains currently in traffic. The map view is updated dynamically, with trains shown as moving circles on the subway map. Both views are clickable: clicking on a station brings up a textual station view in a different window, and similarly, clicking on a train in the train list brings up a textual view with more detailed information about the train.

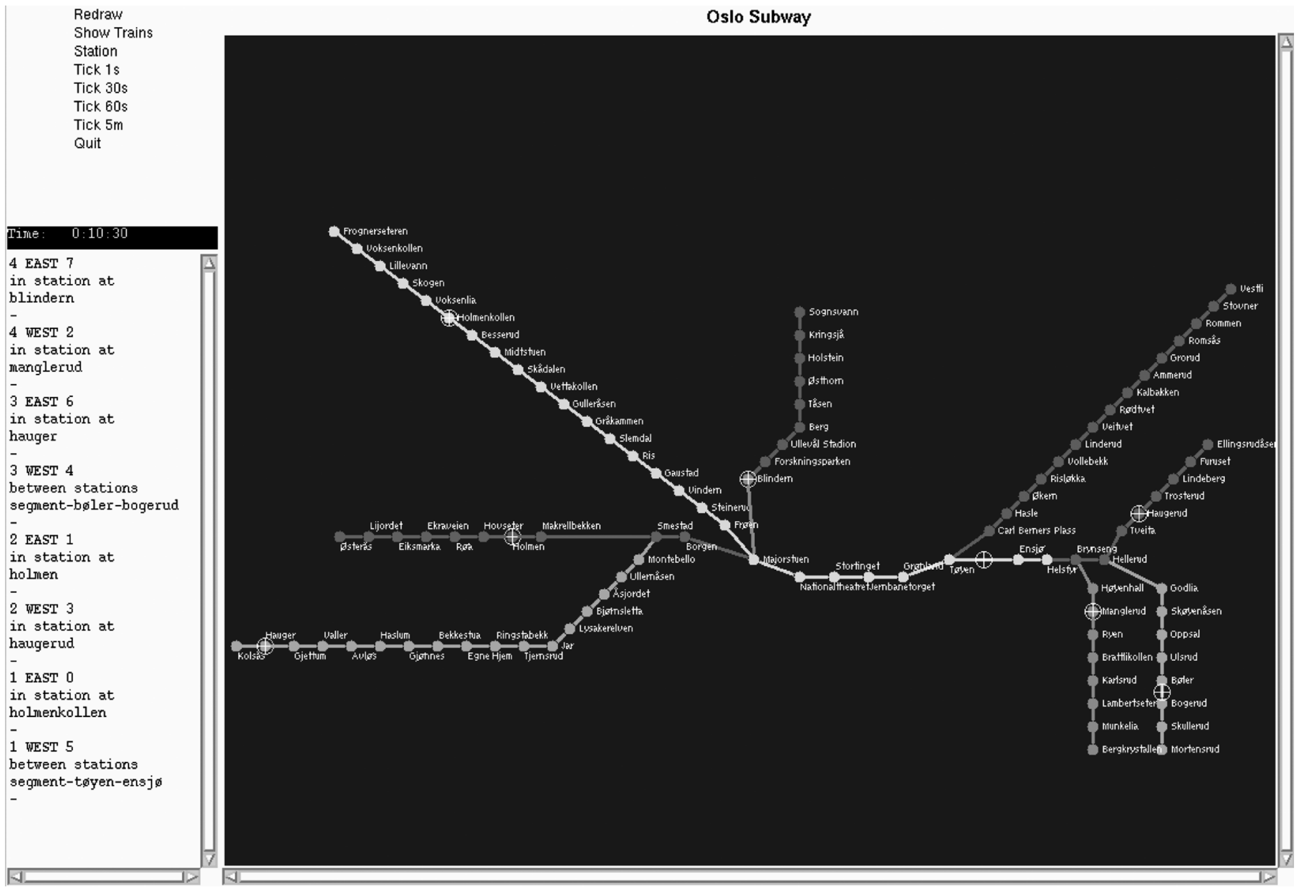
5. DISCUSSION

As noted by van der Aalst [9], and demonstrated more extensively in a number of papers, for instance [10–12], Petri nets can be used to implement the concepts of both logistics and workflow. Our work is inspired by van der Aalst in the way we decompose an application domain into its constituent parts and analyse it in a precise way using high-level Petri nets. With respect to views, there are some ideas that seem to be overlapping in the work of Basten and van der Aalst [13].

The entire subway net totalled around 12 000 parts (places, transitions, and arcs), but we were still able to simulate it faster than real time on a 2.5 GHz Pentium 4 machine. Our current implementation of views is dynamic in the sense that if we, for instance, delete a station object from the Petri net in the editor, that station will disappear from the map, or if we change its position data, it will be relocated on the map. However, it is static in the sense that redefining how train tokens are detected, or changing the shape of a station, for instance, has to be done in code.

5.1. Related work

There has not been much interest in the Petri net community in representing railroad systems in Petri nets except some work described in [14]. The authors present a system written in Design/CPN for controlling a model train system. Every train is equipped with a travelling plan that is locally synchronized with the plans of the other trains. Our work differs from theirs in that we provide techniques and tools for constructing executable models of real railroad systems, used for simulation and control. We separate the various aspects of the model, like the railroad net, the signalling system, and the behaviour at the station.



```
Time:    0:10:30
4 EAST 7
in station at
blindern
-
4 WEST 2
in station at
manglerud
-
3 EAST 6
in station at
hauger
-
3 WEST 4
between stations
segment-bøler-bogerud
-
2 EAST 1
in station at
holmen
-
2 WEST 3
in station at
haugerud
-
1 EAST 0
in station at
holmenkollen
-
1 WEST 5
between stations
segment-tøyen-ensjø
-
```

Fig. 8. The Oslo subway application.

Our Petri net tool has some features that overlap with Design/CPN[3], Renew[4], and Jfern[5]. In [15] various techniques for obtaining information from a simulation are discussed. The concept of view weakly overlaps with the concept of "dashboard object" in ExSpect, or "business chart" in Design/CPN, although neither seems to provide the full generality and flexibility of views.

5.2. Future work

Since we report work in progress, there are several remaining tasks. A formalization of views would be important to clarify the concept and indicate how one can build tool support for constructing views. To use events from the external environment as input, and save histories of executions, and then use these histories to simulate behaviour with the external environment in a faithful way, is something not investigated yet. As noted previously, issues of safety have not been addressed thoroughly in the paper. A solution to this would be to construct a library of railroad components for collision detection and collision alarms, and investigate how we can make use of the verification and formal analysis techniques that Petri nets provide.

6. CONCLUSIONS

A prototype of Andromeda has been implemented, on which we can carry out simulations and some simple performance analysis. Like any Petri net based system, the prototype supports formal analysis [6] and systematic treatment of performance analysis [16], although we have not built tools for advanced analysis yet. A set of views is implemented on top of the net. The main view shows the subway map with trains moving.

Object-orientation turned out to be useful while constructing railroad components, and the inheritance mechanism applied to nets reduced the time spent on modelling railroad components, since we used variants of Petri net patterns several times by subclassing components, specializing them for various purposes.

This project has demonstrated to us how to exploit the best in the two worlds of formal methods and state of the art computing power, by taking a well understood and classic theory, in our case CPNs, and making use of them as an integral part of a running practical application. We see the goal of our work to show that formal methods can play the key role, in building and reasoning about systems of practical importance.

---

[3] The canonical implementation of CPNs, see http://www.daimi.au.dk/designCPN/.

[4] An object-oriented Petri net tool, where Petri nets can implement methods and can be treated as first-class Java objects and Java code can be accessed from nets in Renew http://www.informatik.uni-hamburg.de/TGI/renew/.

[5] A Java-based light weight Petri net tool, see http://sourceforge.net/projects/jfern/.

ACKNOWLEDGEMENTS

The Andromeda project was funded by the Norwegian Computing Center. Thanks go to Yuri Gurevich, Olaf Owe, Bjarte M. Østvold, Arild Torjusen, Einar B. Johnsen, Bent Foyn, Wolfgang Leister, Pål Enger, Ingrid Yu, and Joackim Bjørk for comments and discussions on earlier drafts of the paper. The authors would also like to thank an anonymous referee for very useful comments on the final version of the paper.

REFERENCES

1. Kristoffersen, T., Moen, A. and Hansen, H. A. Simulating the Oslo subway by hierarchic, coloured, object-oriented, timed Petri nets with viewpoints. In *Abstracts from 14th Nordic Workshop on Programming Theory, NWPT'02 (Tallinn, November 2002)* (Vain, J. and Uustalu, T., eds.). Institute of Cybernetics, Tallinn, 2002, 57–60.
2. Petri, C. A. Kommunikation mit Automaten. Schriften des IIM Nr. 2, Institut für Instrumentelle Mathematik, 1962.
3. Jensen, K. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Vol. 1: Basic Concepts*. 2nd ed. Monographs in Theoretical Computer Science: An EATCS Series. Springer-Verlag, Berlin, 1997.
4. Reisig, W. and Rozenberg, G. (eds.). Lectures on Petri Nets: Advances in Petri Nets, Vol. 1: Basic Models. *LNCS*, 1998, **1491**.
5. Reisig, W. and Rozenberg, G. (eds.). Lectures on Petri Nets II: Advances in Petri Nets, Vol. 2: Applications. *LNCS*, 1998, **1492**.
6. Jensen, K. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Vol. 2: Analysis Methods*. 2nd ed. Monographs in Theoretical Computer Science: An EATCS Series. Springer-Verlag, Berlin, 1997.
7. Jensen, K. *Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use, Vol. 3: Practical Use*. 2nd ed. Monographs in Theoretical Computer Science: An EATCS Series. Springer-Verlag, Berlin, 1997.
8. Pachl, J. *Railway Operation and Control*. VTD Rail Publishing, 2002.
9. van der Aalst, W. M. P. *Timed Coloured Petri Nets and Their Application to Logistics*. PhD thesis, Eindhoven University of Technology, 1992.
10. van der Aalst, W. M. P., van Hee, K. M. and Houben, G. J. Modelling and analyzing workflow using a Petrinet based approach. In *Proc. of the Second Workshop on Computer-Supported Cooperative Work, Petri Nets and Related Formalisms* (Michelis, G. D., Ellis, C. and Memmi, G., eds.). 1994, 31–50.
11. van der Aalst, W. M. P. Three good reasons for using a Petrinet based workflow management system. In *Information and Process Integration in Enterprises: Rethinking Documents* (Wakayama, T., Kannapan, S., Khoong, C. M., Navathe, S. and Yates, J., eds.). *Kluwer Int. Series Engineering and Comp. Sci.*, 1998, **428**, 161–182.
12. van der Aalst, W. M. P. The application of Petri nets to workflow management. *J. Circuits Systems Computers*, 1998, **8**, 21–66.
13. Basten, T. and van der Aalst, W. M. P. Inheritance of behavior. *J. Logic Algebr. Program.*, 2001, **47**, 47–145.
14. Hielscher, W., Urbszat, L., Reinke, C. and Kluge, W. On modelling train traffic in a model train system. In *Proc. of 1st Workshop and Tutorial on Practical Use of Coloured Petri Nets and Design/CPN (Aarhus, June 1998)* (Jensen, K., ed.). University of Aarhus, Denmark, 1998.

15. Jensen, K. An introduction to the practical use of coloured Petri nets. In *Lectures on Petri Nets: Advances in Petri Nets, Vol. 2: Applications* (Reisig, W. and Rozenberg, G., eds.). *LNCS*, 1998, **1492**, 237–292.
16. Ajmone Marsan, M., Bobbio, A. and Donatelli, S. Petri nets in performance analysis: an introduction. In *Lectures on Petri Nets: Advances in Petri Nets, Vol. 1: Basic Models* (Reisig, W. and Rozenberg, G., eds.). *LNCS*, 1998, **1491**, 211–256.

Kõrgema taseme informatsiooni tuletamine Petri võrkudest raudtee näitel

Thor Kristoffersen, Anders Moen ja Hallstein Asheim Hansen

On kirjeldatud süsteemi modelleerimise metoodikat, mis põhineb objekt-orienteeritud laiendustega värvilistel Petri võrkudel. Metoodika olulisemaid iseärasusi on objekt-orienteeritud teisendusskeem, mida nimetatakse vaateks. Vaated on vajalikud mudeli uurimise all olevate aspektide väljatoomiseks ning kasutajale kuvamiseks ebaolulisi detaile varjates. Metoodika on realiseeritud prototüüptööriistal Andromeda, mis võimaldab süsteeme objekt-orienteeritud laiendustega värviliste Petri võrkudena modelleerida ning teha simulatsioone ja lihtsat jõudlusanalüüsi. Artiklit läbiv näide on Oslo metroo.