University of Oslo
Department of Informatics

Executing Large Scale Colored Petri Nets by Using Maude

Simulating Railroad Systems

Joakim Bjørk

Cand. Scient. Thesis

May 8, 2006

Preface

This thesis is submitted to the Department of Informatics at the University of Oslo as part of the *Candidata scientiarum* (Cand. scient.) degree. The work is carries out at the *Precise modeling and analysis* (PMA) research group.

First I would like to thank my tutor on this thesis, Anders Moen Hagalisletto, for priceless advises and discussions. Without him there would not be any thesis.

I would also like to thank Pål Enger for a good collaboration on the software development, and Ingrid Chieh Yu for giving me full access, and an introduction, to the source code of RWSEditor. Further I would like to thank Thor Georg Sælid and Trygve Kaasa from Oslo Sporveier for sharing their knowledge on the subway system of Oslo. I would also like to thank Johan Dovland for valuable comments and advises in the final state of the process.

Finally I would like to thank Silje N. Berglund for her patience.

Joakim Bjørk
May 8, 2006

Contents

List of Figures

Chapter 1

Introduction

This thesis is part of a larger study of the advantages of using colored Petri nets as a modeling language for railway systems. The experiences from this study are presented several papers ([11], [7], [6]). The papers have focused on the structure of the railroad layout and how various behaviors like safety, collision detection, and sensitivity can be constructed as libraries of colored Petri net components and how these libraries of components are related to each other through a relation of syntactic refinement. The railroad layout serves as a specification layer, and basic railroad components are constructed using colored Petri nets. These basic railroad components corresponds to physical railroad components like turnouts, crossings, slips, etc. Then large scale colored Petri net models of the entire Oslo subway system is generated automatically. The colored Petri nets are then automatically translated to Maude code.

At the moment the standard Petri net tools fail to support industrial applications of railroad models in several ways; editing of nets gets infeasible because of (*i*) the Petri net hierarchies, high level Petri nets does not resemble railroad layouts as used in the industry, (*ii*) the size of nets exceeds the allowed usage of memory, and (*iii*) the execution time for well known tools like Design/CPN[1], Renew[2] are not applicable even for small railroad nets. In this thesis, we will explore the use of Maude as execution environment.

One goal of our research is to address Dines Bjørner's "Grand Challenge

[1]`http://www.daimi.au.dk/designCPN/`

[2]`http://www.renew.de`

on the Railway Domain"[3], by giving generic components and methods for construction, analysis and standardization of notation in the railroad domain [1].

The main work on this thesis consists of the following parts:

1. Development of new refinement classes.
2. Expand the algebra of specifications.
3. Introduce the railroad nets as an extension of colored Petri nets.
4. Look at different ways to translate colored Petri nets into Maude modules.
5. Run simulations of large scale railroads.
6. Software development.
    (a) Add new functionality to the tool (RWSEditor) developed to construct the specifications and generate large scale colored Petri nets.
    (b) Development of a tool for automatic translation of colored Petri nets into Maude modules.
    (c) Make a tool for automatic refinement of railroad nets.

Parts of this thesis are presented:

- as the poster "Timed Petri Nets Represented in Maude" at the 9th Estonian Winter School in Computer Science (EWSCS 2004), Palmse, Estonia.
- as the talk "Challenges in Simulating Railway Systems using Petri Nets" at the The TRain Workshop at the 3rd IEEE International Conference on Software Engineering and Formal Methods (SEFM 2005), Koblenz, Germany.
- in the paper [6]: "Constructing and Refining Large Scale Railroad Models Represented by Petri Nets", accepted for publication in IEEE Transactions on Systems, Man and Cybernetics, Part C.

[3]FMRail at `http://www.railwaydomain.org/`.

- in the paper [5]: “Large Scale Simulations of Railroad Nets”, submitted to the 4th International Workshop on Modeling of Objects, Components, and Agents (MOCA 2006).

The poster, the foils from the talk and the articles may be found on the CD-ROM attached to this thesis.

Overview

This thesis is built up of nine chapters. In Chapter 2 some technical preliminaries are given. The chapter has short introductions to both Maude and colored Petri nets. It is also explained some railway terms and mathematical concepts. Specifications are defined and explained in Chapter 3. Here both the general specification language and the specific language used to model the Oslo subway systems are considered. Chapter 4 has a description and a definition of the railroad nets. These are extensions of colored Petri nets. Models of the atomic railway components represented as railroad nets are also presented here. In Chapter 5 several ways of refining the railroad nets are discussed. The connection between specifications and railroad nets is shown in Chapter 6. This is what we call saturation. The chapter also has a description on how to translate railroad nets into executable Maude modules. Two different approaches are explored. Chapter 7 contains an introduction to the software developed in this project. This includes several programs. The results of the simulations we have done are presented in Chapter 8. Here both computational complexity and data size are discussed. Chapter 9 contains the conclusion of this thesis as well as some suggestions for further work.

Chapter 2

Preliminaries

In this chapter some technical preliminaries are given, including some functions, the definition of colored Petri nets, and a brief explanation of Maude. Some railroad expressions are explained and there is a short presentation of the subway system of Oslo.

2.1 Technical preliminaries

This section contains a summary of the technical prerequisites used in this thesis. A set is an unordered collection of distinguishable elements, and is written $\mathrm{S} = \{\mathrm{e}_1, \ldots, \mathrm{e}_n\}$. The empty set is denoted $\emptyset$, and the union operator is denoted $\cup$. The size of sets are defined in Definition 1.

Definition 1 *If* S *is a finite set then* $\mid \mathrm{S} \mid$ *is the number of elements in* S.

The set subtraction operator ($\backslash$) removes all elements of one set from another.

Definition 2 *The set subtraction operator is defined by:*

$$\mathrm{S} \backslash \mathrm{S}' = \{\mathrm{e} | \mathrm{e} \in \mathrm{S} \wedge \mathrm{e} \notin \mathrm{S}'\},$$

where S *and* S' *are finite sets and* e *is an element.*

We write finite sequences as $\mathrm{X} = \langle \mathrm{e}_1, \ldots, \mathrm{e}_n \rangle$. The empty sequence is denoted $\langle\rangle$. To be able to expand a sequence we need a push-last function. This function is defined in Definition 3.

Definition 3 *The push-last function, $\vdash$, extends a sequence by adding an element at the end, and is defined by the following equation:*

$$\langle \mathrm{e}_1, \ldots, \mathrm{e}_n \rangle \vdash \mathrm{e}_{n+1} = \langle \mathrm{e}_1, \ldots, \mathrm{e}_n, \mathrm{e}_{n+1} \rangle$$

We need a notion of length of finite sequences. This is defined in Definition 4.

Definition 4 *The length of a finite sequence* X *denoted* $\mid \mathrm{X} \mid$ *is defined by:*

1. $\mid \langle\rangle \mid = 0$
2. $\mid \mathrm{X}' \vdash \mathrm{e} \mid = \mid \mathrm{X}' \mid +1$

Sequences of length 1 is called singles, sequences of length 2 are referred to as pairs etc.

Definition 5 *A sequence of n empty sets, denoted $\emptyset_n$ where $n \in \mathbb{N}$ is defined by the following equations:*

1. $\emptyset_1 = \langle \emptyset \rangle$
2. $\emptyset_i = \emptyset_{i-1} \vdash \emptyset$ *if* $i > 1$

Multi-sets can as opposed to sets contain more than one indistinguishable element. Elements of a multi-set are, as is the case for sets, unordered.

Definition 6 *A multi-set* Z *is a pair* $\mathrm{Z} = \langle \mathrm{S}, f \rangle$*, where* S *is a set and* f *is a function* $f : \mathrm{S} \mapsto \mathbb{N}$*. The size of a multi-set* Z*, denoted* $\mid \mathrm{Z} \mid$*, is given by* $\sum_{\mathrm{e} \in \mathrm{S}} f(\mathrm{e})$.

Definition 7 *A* monoid $\mathcal{M}$ *is a pair* $\langle \mathrm{S}, * \rangle$*, where* S *is a set, and* $*$ *is a binary operator that satisfies the following laws:*

*1. **Closure.** For all elements* $\mathrm{e}_1, \mathrm{e}_2 \in \mathrm{S}$*,*

$$\mathrm{e}_1 * \mathrm{e}_2 \in \mathrm{S}$$

*2. **Identity.** There exists an element* $\boldsymbol{e} \in \mathrm{S}$ *such that, for all* $\mathrm{e}_2 \in \mathrm{S}$*,*

$$\boldsymbol{e} * \mathrm{e}_2 = \mathrm{e}_2 = \mathrm{e}_2 * \boldsymbol{e}$$

*3. **Associativity.** For all elements* $\mathrm{e}_1, \mathrm{e}_2, \mathrm{e}_3 \in \mathrm{S}$*,*

$$\mathrm{e}_1 * (\mathrm{e}_2 * \mathrm{e}_3) = (\mathrm{e}_1 * \mathrm{e}_2) * \mathrm{e}_3$$

Definition 8 *A monoid* $\mathcal{M} = \langle \mathrm{S}, * \rangle$*, is an abelian monoid (or commutative monoid) if the following law is satisfied:*

*4. **Commutativity.** For all elements* $\mathrm{e}_1, \mathrm{e}_2 \in \mathrm{S}$*,*

$$\mathrm{e}_1 * \mathrm{e}_2 = \mathrm{e}_2 * \mathrm{e}_1$$

Definition 9 *The flat union, denoted* $\sqcup$*, of two finite sequences of equal length* $\mathrm{X} = \langle \mathrm{e}_1, \ldots, \mathrm{e}_n \rangle$ *and* $\mathrm{X}' = \langle \mathrm{e}'_1, \ldots, \mathrm{e}'_n \rangle$*, given by* $\mathrm{X} \sqcup \mathrm{X}' = \langle \mathrm{e}_1 \cup \mathrm{e}'_1, \ldots, \mathrm{e}_n \cup \mathrm{e}'_n \rangle$*.*

Lemma 1 *The set of all possible sequences of length* n*, where* $n \in \mathbb{N}$ *and all element are sets, and the flat union of these sequences, is an abelian monoid.*

Proof: Let $\mathrm{X} = \langle \mathrm{e}_1, \ldots, \mathrm{e}_n \rangle$, $\mathrm{X}' = \langle \mathrm{e}'_1, \ldots, \mathrm{e}'_n \rangle$ and $\mathrm{X}'' = \langle \mathrm{e}''_1, \ldots, \mathrm{e}''_n \rangle$ be sequences of length n, where all elements e_i, e'_i and e''_i are sets.
Closure: We must prove that for all X and X′, $\mathrm{X} \sqcup \mathrm{X}'$ are a sequence of length n, where all elements are sets. $\mathrm{X} \sqcup \mathrm{X}' = \langle \mathrm{e}_1 \cup \mathrm{e}'_1, \ldots, \mathrm{e}_n \cup \mathrm{e}'_n \rangle$, which obviously is a sequence of n elements. All these elements are sets, since the union of two sets is a set. The closure property hold for the flat union since it holds for the union of sets.
Identity: We must prove that there exists a sequence, $\emptyset_n$, of length n such that $\mathrm{X} \sqcup \emptyset_n = \mathrm{X} = \emptyset_n \sqcup \mathrm{X}$. $\emptyset_n$ is the sequence of n empty sets. Then $\mathrm{X} \sqcup \emptyset_n = \langle \mathrm{e}_1 \cup \emptyset, \ldots, \mathrm{e}_n \cup \emptyset \rangle = \langle \mathrm{e}_1, \ldots, \mathrm{e}_n \rangle = \mathrm{X}$, and $\emptyset_n \sqcup \mathrm{X} = \langle \emptyset \cup \mathrm{e}_1, \ldots, \emptyset \cup \mathrm{e}_n \rangle = \mathrm{X}$.

Associativity: Then we must show that $\mathrm{X} \sqcup (\mathrm{X}' \sqcup \mathrm{X}'') = (\mathrm{X} \sqcup \mathrm{X}') \sqcup \mathrm{X}''$.

$$\begin{aligned}
\mathrm{X} \sqcup (\mathrm{X}' \sqcup \mathrm{X}'') &= \mathrm{X} \sqcup \langle \mathrm{e}_1' \cup \mathrm{e}_1'', \ldots, \mathrm{e}_n' \cup \mathrm{e}_n'' \rangle \\
&= \langle \mathrm{e}_1 \cup (\mathrm{e}_1' \cup \mathrm{e}_1''), \ldots, \mathrm{e}_1 \cup (\mathrm{e}_n' \cup \mathrm{e}_n'') \rangle \\
&\overset{1}{=} \langle (\mathrm{e}_1 \cup \mathrm{e}_1') \cup \mathrm{e}_1'', \ldots, (\mathrm{e}_n \cup \mathrm{e}_n') \cup \mathrm{e}_n'' \rangle \\
&= \langle \mathrm{e}_1 \cup \mathrm{e}_1', \ldots, \mathrm{e}_n \cup \mathrm{e}_n' \rangle \sqcup \mathrm{X}'' \\
&= (\mathrm{X} \sqcup \mathrm{X}') \sqcup \mathrm{X}''
\end{aligned}$$

Equation 1 is valid because the union operator, $(\cup)$, is associative.
Commutativity: Finally we must prove that $\mathrm{X} \sqcup \mathrm{X}' = \mathrm{X}' \sqcup \mathrm{X}$.

$$\begin{aligned}
\mathrm{X} \sqcup \mathrm{X}' &= \langle \mathrm{e}_1 \cup \mathrm{e}_1', \ldots, \mathrm{e}_n \cup \mathrm{e}_n' \rangle \\
&\overset{2}{=} \langle \mathrm{e}_1' \cup \mathrm{e}_1, \ldots, \mathrm{e}_n' \cup \mathrm{e}_n \rangle \\
&= \mathrm{X}' \sqcup \mathrm{X}
\end{aligned}$$

Note that Equation 2 is valid since the union operator is commutative.

The flat union is an abelian monoid since the union of sets is an abelian monoid. ♣

A partial monoid as proposed by Anders Moen Hagalisletto in [6] is defined as follows:

Definition 10 *A partial monoid is a 4-tuple* $\langle \mathrm{S}, \star, \mathbb{B}, \mathcal{J} \rangle$*, where* S *is a set,* $\star$ *is a set of parameterized binary operators,* $\star = \langle \star_b | b \in \mathbb{B} \rangle$*,* $\mathbb{B}$ *is a connected set of pairs, and* $\mathcal{J}$ *(joinable) is a ternary relation over* $\mathrm{S} \times \mathrm{S} \times \mathbb{B} \mapsto \mathrm{Bool}$*, such that:*

1. *If* $\mathrm{e}_1, \mathrm{e}_2 \in \mathrm{S}$ *and* $\mathcal{J}(\mathrm{e}_1, \mathrm{e}_2, b)$ *then* $\mathrm{e}_1 \star_b \mathrm{e}_2 \in \mathrm{S}$
2. $(\mathrm{e}_1 \star_{b_1} \mathrm{e}_2) \star_{b_2} \mathrm{e}_3 = \mathrm{e}_1 \star_{b_1} (\mathrm{e}_2 \star_{b_2} \mathrm{e}_3)$
3. $\exists\, \mathrm{e}_1 \in \mathrm{S}\, \exists b \in \mathbb{B}\, \forall \mathrm{e}_2 \in \mathrm{S}\ \mathrm{e}_2 \star_b \mathrm{e}_1 = \mathrm{e}_2 = \mathrm{e}_1 \star_b \mathrm{e}_2$

Definition 11 *A partial monoid* $\langle \mathrm{S}, \star, \mathbb{B}, \mathcal{J} \rangle$ *is a partial abelian monoid if it satisfies the law of commutativity.*

4. $\mathrm{e}_1 \star_b \mathrm{e}_2 = \mathrm{e}_2 \star_b \mathrm{e}_1$

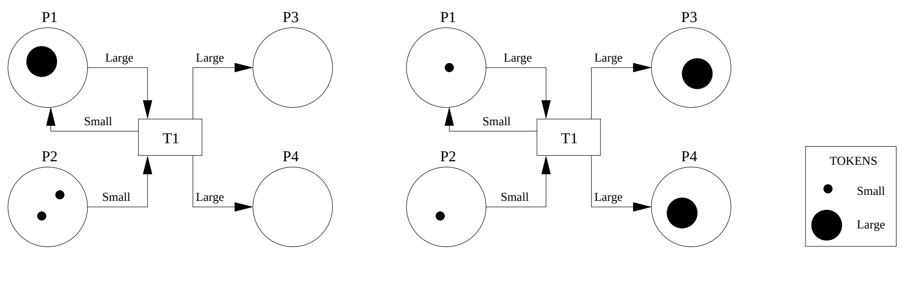
Definition 12 *Let $\circ$ be the unordered concatenation function between elements and be defined as:* $\mathrm{e}_1 \circ \mathrm{e}_2 = \mathrm{e}_1 \circ \mathrm{e}_2$

Definition 13 *Let π_i be the projection function for sequences such that:*

1. $\pi_1(\langle \mathrm{e}_1, \ldots, \mathrm{e}_n \rangle) = \mathrm{e}_1$
2. $\pi_i(\langle \mathrm{e}_1, \mathrm{e}_2, \ldots, \mathrm{e}_n \rangle) = \pi_{i-1}(\langle \mathrm{e}_2, \ldots, \mathrm{e}_n \rangle)$ *if* $i > 1$
3. $\pi_i(\langle\rangle) = \bot, \forall i \in \mathbb{N}$

2.2 Petri nets

Petri net is a graphical programming language introduced by Carl Adam Petri in 1962 [18], and has evolved into many dialects. We shall now describe a limited version of colored Petri nets (CPN) (see [9], [10]) applied to railway modeling. We will first describe CPN by a small example, and then go through a formal definition. A complete definition of colored Petri nets is not required, for a careful introduction to CPN see for instance [9].



(a) Initial marking (b) After firing of T1

Figure 2.1: Firing of a transition.

Figure 2.1(a) shows a small colored Petri net. The four large circles are *places*, and the rectangle is a *transition*. The transitions performs the changes in the system. The filled circles inside some of the places are *tokens*. There are two kinds of tokens in the figure, small and large. The different kinds of tokens are referred to as colors. In uncolored Petri nets, all tokens are indistinguishable. The distribution of tokens over the places is called a *marking*. This is the state of the system. The marking of a Petri net before any actions have occurred is called the *initial marking*. All the arrows are called *arcs*. The text above the arcs are *arc expressions*. The arcs leads tokens either from a place to a transition or vice versa. The input places of a transition are all places that have an arc to the transition. Likewise the output places of a transition are all places with an arc from a transition. Note that a place can be both an output place and an input place of the same transition. In Figure 2.1(a) the place `P1` is both an input place, and an output place of the transition `T1`. A transition is said to be *enabled* if all arcs that leads to this transition may lead a token according to the arc expression. In Figure 2.1(a) the conditions for the transition `T1` to be enabled is that there is a large token in `P1` and a small token in `P2`. This is the case so `T1` is enabled. An enabled transition may *fire*. When a transition fires tokens are consumed from its input places and new tokens are added to its output places according to the arc expressions. If the transition `T1` in Figure 2.1(a) fires, then one large token is removed from `P1` and a small token is removed from `P2`. At the same time a large token is added to each of the places `P3` and `P4`, and a small token is added to the place `P1`. We will then have the situation of Figure 2.1(b). Now `T1` is no longer enabled because there are no large tokens in `P1`.

Definition 14 *An unmarked colored Petri net* CPN *is a triple,* $\langle P, T, A\rangle$, *where*

1. P *is a finite set of places.*
2. T *is a finite set of transitions.*
3. A *is a finite set of arcs,* $A \subseteq (P \times T \times E) \cup (T \times P \times E)$.

E denotes a set of expressions that can be boolean tests, assignments and time inscriptions. Boolean tests are written $t = x$, or just x for short, assignments are written $t := x$, while delays are denoted $@ + x$. The empty expression is denoted by **nil**, and is usually omitted. Subexpression is defined by recursion: Every expression e is a subexpression of itself, **nil** is a subexpression of every expression, e is a subexpression of both $e \wedge e'$ and $e' \wedge e$, where e' is an expression. A *node* in a colored Petri net is either a place or a transition, i.e x is a node if $x \in P \cup T$, while arcs connect places and transitions. A transition is a triple $\langle \mathrm{TransId}, \mathrm{Name}, \mathrm{Expr} \rangle$, where TransId is its unique identity, Name its name and Expr is an expression, also called *guard* in [9]. A place is a pair $\langle \mathrm{PlaceId}, \mathrm{Color} \rangle$, that contains a unique identity and a color. The empty colored Petri net $\mathrm{CPN}_\emptyset$ is defined as $\langle \emptyset, \emptyset, \emptyset \rangle$.

Let $\mathscr{C}$ denote a finite set of colors, $\mathscr{C} = \{\mathscr{C}_1, \ldots, \mathscr{C}_n\}$.

Definition 15 *A colored token is a pair, $\langle \mathscr{C}_i, \mathrm{atr} \rangle$, where*

1. *$\mathscr{C}_i$ is the color of the token, such that $\mathscr{C}_i \in \mathscr{C}$.*
2. atr *is a sequence of color specific attributes.*

Note that atr may be empty. Tokens might inhabit places. The distribution of tokens over places gives a *marking*. Let $\mathbb{M}$ denote the set of all possible multi-sets of tokens. Formally a marking is a set of pairs of places and multi-sets of tokens, $\{\langle p, \mathrm{m} \rangle | p \in P \wedge \mathrm{m} \in \mathbb{M}\}$. Given a marking M then we define $\mathbb{m}(p, \mathrm{M}) = \langle p, \mathrm{m} \rangle$ such that $\langle p, \mathrm{m} \rangle \in \mathrm{M}$, if there is any, otherwise it is undefined. The initial marking is denoted $\mathrm{M}_{\mathrm{init}}$. We say that a Petri net is *1-safe* if it never can reach a marking where a place contains more than one token from an initial marking where no places contains more than one token. $| \pi_2(\mathbb{m}(p, \mathrm{M}_{\mathrm{init}})) | \leq 1, \forall p \in P \rightarrow | \pi_2(\mathbb{m}(p, \mathrm{M})) | \leq 1, \forall p \in P$ for all reachable markings M from $\mathrm{M}_{\mathrm{init}}$. The *occurrence graph* is a graph of all possible markings reachable from a given initial marking.

The *preset* of the transition t, denoted ${}^\bullet t$, is the set of input places of t. The *postset* of the transition, denoted $t^\bullet$ is the set of output places of t.

2.2.1 Timed models

When modeling real life systems, time is often essential. This is particularly the case for railroad modeling. Introducing time enables us to reason about many problems. Examples of such problems are:

- How long time does it take for a train to drive a given stretch.
- Whether a train is able to follow its timetable.

- How delays influences the other trains due to track sharing.
- How to make the time tables as efficient as possible.

Concepts of time

Even the simplest Petri nets have a notion of time. Even though the net does not count them we do talk about execution steps. These steps are some kind of time units and they are essential to talk about the ordering of events. In the net of Figure 2.2 it is obvious that transition T1 must occur before T2. This also makes it possible to reason about other qualitative temporal properties like liveness, dead-locks, fairness, etc.

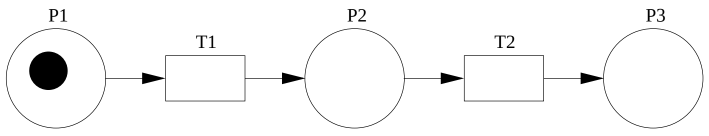


Figure 2.2: Order of firing

In order to use the Petri net framework to model real time systems it would be useful to quantify the time units. This can be done in High Leveled Petri nets. Quantifying the time units will make it possible to reason about duration, delays, deadlines, etc. There have been many different approaches to introduce time to colored Petri nets.

Discrete or continuous time

Is time really a continuous or a discrete scale? The most common conclusion is probably that time is continuous. We can not divide time into atomic units. All our instruments used to measure time must be discrete though. We can not build a clock with infinite precision. It will always have a finite number of decimal places and therefore be rounded. A computer operates with discrete time, with a clock cycle as an atomic time unit.

So even though a discrete time scale might be the most natural choice, models often use a continuous time scale. This is to avoid the problem with multiple events occurring between two time stamps and having to consider all possible orders of them. With a continuous time scale all atomic events can be ordered in a strictly monotone order [13]. $t(e_1) < t(e_2)$ or $t(e_1) > t(e_2)$ for all atomic events e_1 and e_2 where t(e) is the time where e occurs.

We have chosen discrete time for all models in this thesis. We have used one second as the smallest time unit. This is small enough for our models, but can obviously be even smaller if it should be necessary.

Location of time delays

One way of handling delays is to add a delay to the transitions. In these kind of nets a transition must remain enabled for a specific time before it can occur [22]. There are two ways of handling what should happen if a transition becomes unabled after being enabled for a time. One could either reset the transitions counter and let it start over the next time it becomes enabled or let it remember how long it has been enabled for and let it start from there.

Another way of handling the delays is to let some time pass from the tokens disappear from the transitions input places to the time they appear in the output places. This is refereed to as *two-phase* firing. Two-phase firing generates a problem if we want to create a occurrence graph. How should we handle the tokens that are in transition. One solution to this is to add tokens with time stamps to the output places. This timestamp indicates at what time the token is ready. They can not be consumed unless the time of a global clock is equal to or larger than the time stamp, as explained in [9]. All timed models used in this thesis have two-phase firing with time stamps added to the delayed tokens.

Kinds of delays

The time delays may be ordered into three groups as described in [22]. The three types are *stochastic* delays, delays specified by *intervals*, and *deterministic* delays.

Stochastic delays
The firing time of a transition is determined by a stochastic process. Stochastic delays will be exponentially distributed.

Delays specified by intervals
Each transition has assigned a maximum and a minimum delay to it. The transition lasts for a random amount of time between the maximum and the minimum boundary.

Deterministic delays
Deterministic delays may either be fixed, or calculated by a function. With fixed delays each firing of a specific transition lasts for a fixed time. The weakness of this approach is that in real life situations several applications of the same action usually do not take exactly the same amount of time. It may be influenced by external factors. The time it takes for a train to move from point a to point b may vary as a result of the weather conditions, who is driving the train etc. If we want to let the delay be dependent on such factors, we must define a function to calculate it. All delays used in this thesis are deterministic.

2.3 Rewriting Logic and Maude

In rewriting logic the basic axioms are rewriting rules of the form $l : \mathrm{t} \rightarrow \mathrm{t}'$, where t and t′ are expressions in a given language, and l is a label. There are two complimentary readings of a rewrite rule $l : \mathrm{t} \rightarrow \mathrm{t}'$, one computational and another logical. For more details see [14].

- Computationally the rewrite rule $l : \mathrm{t} \rightarrow \mathrm{t}'$ is interpreted as a local transition in a concurrent system; that is, t and t′ describe patterns for fragments of the distributed state of a system, and the rule explain how a local concurrent transition can take place in such a system, changing the local state fragment from an instance of the pattern t to the corresponding instance of the form t′. The rest of the system remains unchanged.
- Logically the rewrite rule $l : \mathrm{t} \rightarrow \mathrm{t}'$ is interpreted as an inference rule, so that we can infer formulas of the form t′ from formulas of the form t.

Definition 16 *A rewrite theory is a 4-tuple* $\mathcal{R} = (\Sigma, \mathrm{E}, \mathrm{L}, R)$ *where:*

1. Σ *is an equational signature.*

2. E *is a set of* $\Sigma -$ *equations.*

3. L *is a set of labels.*

4. R *is a set of labeled rules*[1]*.*

Then (Σ, E) *is the equational theory for rewriting in.*

The definition is from [14, 15].

As a consequence, the relevant sentences that may or may not be provable by the theory $\mathcal{R}$ are sequents of the form $[t]_{\mathrm{E}} \to [t']_{\mathrm{E}}$ where t and t' are Σ-terms, possibly involving some variables, and $[t]_{\mathrm{E}}$ denotes the equivalence classes of the term t modulo the equations E. The provable sentences are exactly those derivable by the following inference rules.

Reflexivity. For each $[t] \in T_{\Sigma,\mathrm{E}}(X)$,

$$\frac{}{[t] \to [t]}$$

Congruence. For each $f \in \Sigma_n, n \in \mathbb{N}$

$$\frac{[t_1] \to [t_1'] \ldots [t_n] \to [t_n']}{[f(t_1, \ldots, t_n)] \to [f(t_1', \ldots, t_n')]}$$

Replacement. For each rule $l : [t(x_1, \ldots, x_n)] \to [t'(x_1, \ldots, x_n)]$ in $\mathcal{R}$,

$$\frac{[w_1] \to [w_1'] \ldots [w_n] \to [w_n']}{[t(\frac{\overline{w}}{\overline{x}})] \to [t'(\frac{\overline{w'}}{\overline{x}})]},$$

where $t(\frac{\overline{w}}{\overline{x}})$ denotes the simultaneous substitution of w_i for x_i in t.

Transitivity.

$$\frac{[t_1] \to [t_2][t_2] \to [t_3]}{[t_1] \to [t_3]}$$

[1]For simplicity we will assume that the rules are unconditional.

Maude is a functional language based on rewriting logic. Rewriting logic is about changing of states. It is therefore particularly well suited to express state-changing aspects of systems [2].

A typical Maude program contains a set of functions and a set of equations and rewriting rules. If the predefined sorts are not adequate, new ones can be defined. It may also contain variables of different sorts. In Maude all functions are declared recursively by equations. Maude has no pointers or aliasing and thereby no side-effects. This makes it possible to prove certain qualities by proving them for each equation without looking at the entire system.

The execution tool for Maude modules has two ways of executing the modules. One way is by using the *rewrite* command, which applies the rules by a top down strategy. The other option is to use the *frewrite* or fair rewrite. This command applies the rules by a position strategy. The Maude tool does not only execute Maude modules. It is also a powerful analysis tool. It has both search capabilities, and a built in model checker.

2.4 Railway systems

In this section some railway expressions used later in this thesis is presented. Figure 2.3 shows a railroad track as seen from above. It consists of two

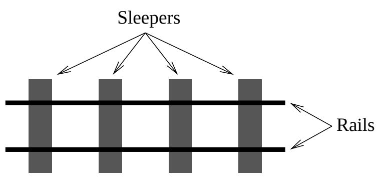


Figure 2.3: Railroad track

parallel *rails* that rests upon *sleepers*. *Movable points* or just *points* are movable rails which can guide the wheels of a train towards one diverging track or the other of a branching component. The point where two rails cross is referred to as a *frog*. *Block sections* is a way to divide the tracks into smaller parts. Block sections may either be fixed or floating. The purpose of such a division is to ensure train separation. A block section is unoccupied if there are no trains in it, otherwise it it occupied. Generally a train may enter a block section only if it is unoccupied. Oslo Sporveier has a slightly different approach. Trains can enter occupied block sections, but at a very low speed. Block sections are used to determine the speed limits of the trains. Generally the more unoccupied block sections there are in front of the train the higher speed limit. For a more thoroughly introduction to railway systems see [17].

2.5 Oslo Sporveier

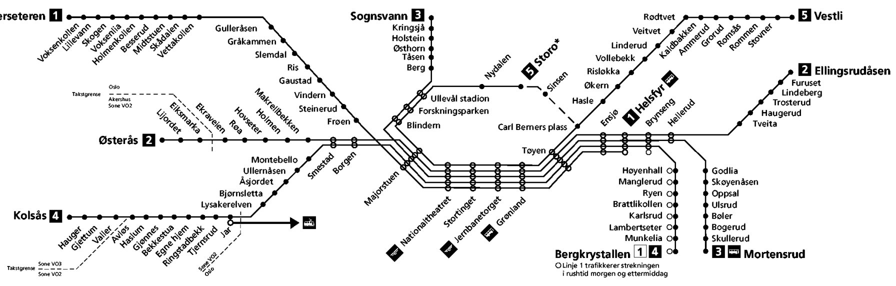


Figure 2.4: Oslo subway system

Oslo Sporveier[2] is the only subway company operating in Oslo. It has 5 lines with a total of 84 kilometers of tracks and 103 stations. It has a total of 59.4 million travelers pr year, which is 36% of all public transport i Oslo[3].

Figure 2.4 shows the current map of the subway of Oslo. From the point of view of simulating the railroad, the size of the data structures and the time it takes to run several trains put hard requirements on the underlying implementation and the hardware, as described in Section 8.1.

The Figure 2.5 shows the specification of the subway of downtown Oslo, the track from Majorstuen to Grønland. This part of the line is very important and safety critical, since trains from every line drive on the same track under ground. Traffic jams are likely to happen, since the interval between the trains can be as low as 90 seconds. If an accident occur, there will soon be a queue of trains influencing the throughput on the complete subway system. The net in Figure 2.5 consists of 93 components, ordinary track segments,

[2]`http://www.sporveien.no/`

[3]The numbers are from 2004

turnouts, end segments, crossings and other components (for more details see [7]).

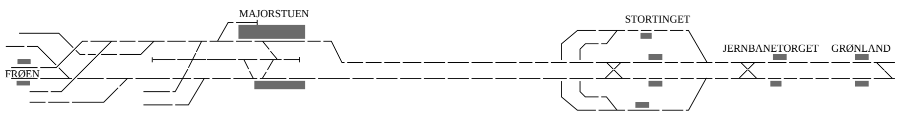


Figure 2.5: The downtown fragment of the subway of Oslo.

Chapter 3

Railroad specification languages

When modeling a railroad topography it is useful to leave out all the details as long as possible. The specifications have appeared to be a suitable abstraction level. A specification can be represented graphically, and will then look like the example in Figure 3.1. Graphs like these are very much alike the ones railroad engineers use, like the one in Figure 2.5. The line maps that passengers use, like the one in Figure 2.4, are not to far from the specification graphs either. This makes the specification graphs understandable to a large variety of people. The specifications presented here are expanded version of the ones presented in [7].

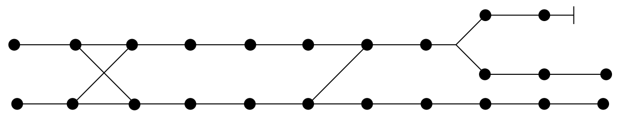


Figure 3.1: Specification of a railroad net.

3.1 The general specification language

A specification language is a set of small graphs, called the template graphs, and instructions on how copies of them can be connected.

Definition 17 *A specification language* $\mathrm{S^L}$ *is a tuple* $\langle \mathrm{G^T}, \mathrm{D}, \mathrm{T}, \mathrm{C}, \mathrm{E} \rangle$.

1. $\mathrm{G^T}$ *is the template graphs, a nonempty, finite set of specification-graphs.*
2. D *is a finite set of non-empty directions.*
3. T *is a finite set of non-empty types.*
4. C *is a nonempty, finite set of combination rules such that* $\mathrm{C} \subseteq \mathrm{D} \times \mathrm{D}$.
5. E *is a finite set of structural exceptions such that* $\mathrm{E} \subseteq \mathrm{T} \times \mathrm{T}$.

The template graphs are representing the building blocks of a railroad system. For an example of a set of such template graphs see Section 3.3.3. All the *interface nodes* (see Definition 19) of the template graphs are assigned to a type and a direction. These types and directions are defined in T and D. The types of the nodes are present to be able to put restrictions on the people designing the specification. The structural exceptions are pairs of types. If the types of two nodes form a pair in the set of structural exceptions, these nodes can not be connected. The direction of a node tells us what direction a train that enters this node is traveling in. The combination rules indicates which directions that can be connected. We must be careful when designing these rules. A carelessly designed combination rule may result in *direction traps*. A direction trap is a point of a specification where trains

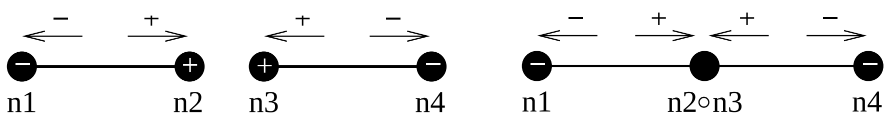


Figure 3.2: Direction trap.

can not move along the intended path due to its direction. Figure 3.2 shows the creation of a direction trap. If the two nodes `n1` and `n2` are connected a direction trap arises. In the new node `n1`∘`n2` trains that enters in the positive direction can not leave due to the direction. The solution to this problem involves some special components called *direction converters* and are further discussed in Section 3.3.3. Note that the name of the new node is the names of the original nodes concatenated by the unordered concatenation function.

The simplest nodes of a specification graph are the internal nodes. These are unconnectable with any other nodes and have no type or direction assigned to them.

Definition 18 *A internal node* $\mathrm{n}^{\mathrm{Int}} = \langle \mathrm{id} \rangle$ *has no attributes except a globally unique identifier.*

Interface nodes are typically present at the end of components and may be connectable to other interface nodes. An interface node has an unique id, a type and a direction assigned to it. Interface nodes that are connected to other interface nodes will have composite types and directions.

Definition 19 *An interface node* n *is a triple* $\langle \mathrm{id}, \mathrm{t}, \mathrm{d} \rangle$ *defined over a specification language* $\mathrm{S}^{\mathrm{L}} = \langle \mathrm{G}^{\mathrm{T}}, \mathrm{D}, \mathrm{T}, \mathrm{C}, \mathrm{E} \rangle$*.*

1. id *is a globally unique identifier.*
2. t *is the type of the node such that* $\mathrm{t} \in \mathrm{T} \cup (\mathrm{T} \circ \mathrm{T})$*.*
3. d *is the direction of the node such that* $\mathrm{d} \in \mathrm{D} \cup (\mathrm{D} \circ \mathrm{D})$*.*

A specification graph is a set of interface nodes, a set of internal nodes and lines between nodes. Both the template graphs of the specification language and the main specification graph are specification graphs.

Definition 20 *A specification-graph* G *is a tuple* $\langle \mathrm{N}, \mathrm{N}^{\mathrm{Int}}, \mathrm{L} \rangle$*.*

1. N *is a nonempty, finite set of interface nodes.*
2. $\mathrm{N}^{\mathrm{Int}}$ *is a finite set of internal nodes.*
3. L *is a finite set of undirected lines, such that* $\mathrm{L} \subseteq (\mathrm{N} \cup \mathrm{N}^{\mathrm{Int}}) \times (\mathrm{N} \cup \mathrm{N}^{\mathrm{Int}})$*.*

The empty specification graph is denoted $\mathrm{G}_\emptyset = \langle \emptyset, \emptyset, \emptyset \rangle$. That a line is undirected means that if $\langle n_1, n_2 \rangle$ is a line then $\langle n_2, n_1 \rangle$ is a line.

Definition 21 *A specification* S *is a pair* $\langle \mathrm{G}, \mathrm{S}^{\mathrm{L}} \rangle$*, where the specification-graph* G *is built up by the template graphs, and by the rules of* S^{L}*.*

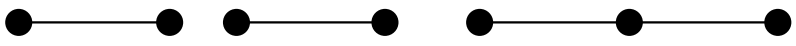
The *empty specification* over a specification language S^{L} is $\mathrm{S}_\emptyset = \langle \mathrm{G}_\emptyset, \mathrm{S}^{\mathrm{L}} \rangle$.

3.2 Composition of specification graphs

Copies of the template graphs can be connected to a larger specification graph in order to model a railway topography. This is done by the joining of interface nodes. In order to do so we need some functions.

First of all, we need a set of projection functions π. π_i extracts the i'th element of a sequence, as defined in Definition 13. If n is an interface node then π_1(n) returns the id of the node, π_2(n) returns the type and π_3(n) returns the direction.

To be able to build larger graphs we must have a way of joining interface nodes. This process is shown in Figure 3.3. The two specification graphs



(a) Two specification graphs

(b) Composite specification graph

Figure 3.3: Composition by joining two interface nodes.

of Figure 3.3(a) can be joined by joining two of the interface nodes. The joining of the two middle interface nodes in Figure 3.3(a), will result in the specification graph in Figure 3.3(b). In order for two interface nodes to be joinable their directions must form a pair in the set of combination rules, and their types must not form a pair in the set of structural exceptions. No interface node is joinable itself.

Definition 22 *Two interface nodes n_1 and n_2, defined in the same specification language $\mathrm{S^L} = \langle \mathrm{G^T}, \mathrm{D}, \mathrm{T}, \mathrm{C}, \mathrm{E} \rangle$, are joinable if:*

1. $\pi_1(n_1) \neq \pi_1(n_2)$, *and*
2. $\langle \pi_2(n_1), \pi_2(n_2) \rangle \notin \mathrm{E}$, *and*
3. $\langle \pi_3(n_1), \pi_3(n_2) \rangle \in \mathrm{C}$

When two joinable interface nodes are joined they merge into a new interface node referred to as a composite node.

Definition 23 *The composition of two joinable interface nodes $n_1 = \langle \mathrm{id}_1, \mathrm{t}_1, \mathrm{d}_1 \rangle$ and $n_2 = \langle \mathrm{id}_2, \mathrm{t}_2, \mathrm{d}_2 \rangle$ written $n_1 \circledn n_2$ is defined as:*

$$n_1 \textcircled{n} n_2 = \langle \mathrm{id}_1 \circ \mathrm{id}_2, \mathrm{t}_1 \circ \mathrm{t}_2, \mathrm{d}_1 \circ \mathrm{d}_2 \rangle$$

When two interface nodes are joined, the direction of the composite new node is always concatenated by $\circ$. Since the combination rules always are pairs of directions, the direction of the new node does not match any combination rule. Therefore a composite node is not joinable with any other interface node.

When replacing an interface node in a specification graph with a new one, not only do the old node have to be replaced, but all lines with an endpoint in the old node must be redirected to the new node.

Definition 24 *The replacement of an interface node n_{old} with a new interface node n_{new} in a specification graph $\mathrm{G} = \langle \mathrm{N}, \mathrm{N^{Int}}, \mathrm{L} \rangle$, is carried out by the substitution function sub.*

1. $sub(n_{old}, n_{new}, \langle \mathrm{N}, \mathrm{N^{Int}}, \mathrm{L} \rangle) =$
   $\langle sub(n_{old}, n_{new}, \mathrm{N}), \mathrm{N^{Int}}, sub(n_{old}, n_{new}, \mathrm{L}) \rangle$
2. $sub(n_{old}, n_{new}, \{n\} \cup \mathrm{N}') = \{n\} \cup sub(n_{old}, n_{new}, \mathrm{N}' \backslash \{n\})$ *if* $n_{old} \neq n$
3. $sub(n_{old}, n_{new}, \{n_{old}\} \cup \mathrm{N}') = \{n_{new}\} \cup \mathrm{N}' \backslash \{n_{old}\}$
4. $sub(n_{old}, n_{new}, \{\langle n, m \rangle\} \cup \mathrm{L}') = \{\langle n, m \rangle\} \cup sub(n_{old}, n_{new}, \mathrm{L}' \backslash \{\langle n, m \rangle\})$
   if $n_{old} \neq n \wedge n_{old} \neq m$

5. $sub(n_{old}, n_{new}, \{\langle n_{old}, m\rangle\} \cup \mathrm{L}') =$
   $\{\langle n_{new}, m\rangle\} \cup sub(n_{old}, n_{new}, \mathrm{L}' \backslash \{\langle n_{old}, m\rangle\})$ *if* $n_{old} \neq m$

6. $sub(n_{old}, n_{new}, \{\langle n, n_{old}\rangle\} \cup \mathrm{L}') =$
   $\{\langle n, n_{new}\rangle\} \cup sub(n_{old}, n_{new}, \mathrm{L}' \backslash \{\langle n, n_{old}\rangle\})$ *if* $n_{old} \neq n$

7. $sub(n_{old}, n_{new}, \{\langle n_{old}, n_{old}\rangle\} \cup \mathrm{L}') =$
   $\{\langle n_{new}, n_{new}\rangle\} \cup sub(n_{old}, n_{new}, \mathrm{L}' \backslash \{\langle n_{old}, n_{old}\rangle\})$

8. $sub(n_{old}, n_{new}, \emptyset) = \emptyset$

where n *and* m *are interface nodes,* L' *is a set of lines and* N' *is a set of nodes.*

Equation 1 in Definition 24 states that to substitute an old node with a new one in a graph is the same as substituting the two nodes in the set of interface nodes, and in the set of lines. The internal nodes will be left unchanged. Equation 2 means that nodes should be left unchanged if they are not equal to the node that are supposed to be substituted. The nodes that are equal to the one that shall be substituted are replaced by the new node in Equation 3. Equation 4 to 7 takes care of the redirections of lines. Equation 4 states that lines, with no endpoints in the node that are supposed to be substituted, are left unchanged. The Equations 4, 5 and 6 redirects lines with one or both endpoints in the node that will be replaced. Equation 8 ends the recursion when a set of either interface nodes or lines are empty.

When two specifications graphs are joined (or a specification graph is joined with itself) it is by the joining of two nodes. If the nodes n_1 and n_2 are joined, they are both replaced by their union node. We then say that the specification graphs are joined with the binding $b = [n_1, n_2]$.

Definition 25 *Two specifications graphs* G_1 *and* G_2 *defined in the same specification language* $(\mathrm{S}^{\mathrm{L}})$*, are joinable with the binding* $b = [n_1, n_2]$ *if* n_1 *is an interface node of* G_1*,* n_2 *is an interface node of* G_2*, and if* n_1 *and* n_2 *are joinable.*

Definition 26 *Let* G_1 *and* G_2 *be two joinable specification graphs over the binding* $b = [n_1, n_2]$*, where* n_1 *is an interface node in* G_1*, and* n_2 *is an interface node in* G_2*. Then the composition of* G_1 *and* G_2 *is given by:*

$$\mathrm{G}_1 \sqcap_b \mathrm{G}_2 = sub(n_2, n_1 \textcircled{n} n_2, sub(n_1, n_1 \textcircled{n} n_2, (\mathrm{G}_1 \sqcup \mathrm{G}_2)))$$

Note that a specification graph may be joinable with itself. If G_1 and G_2 are the same graph, then n_1 and n_2 must be replaced in both G_1 and G_2. If not the composition will result in extra copies of the nodes n_1 and n_2. On the other hand if G_1 and G_2 are two different graphs, there is no harm in replacing both n_1 and n_2 in both graphs. There is no effect of replacing a node that does not exist. Connecting a graph to itself is necessary when modeling a loop. Figure 3.4 shows this process. In the beginning we can

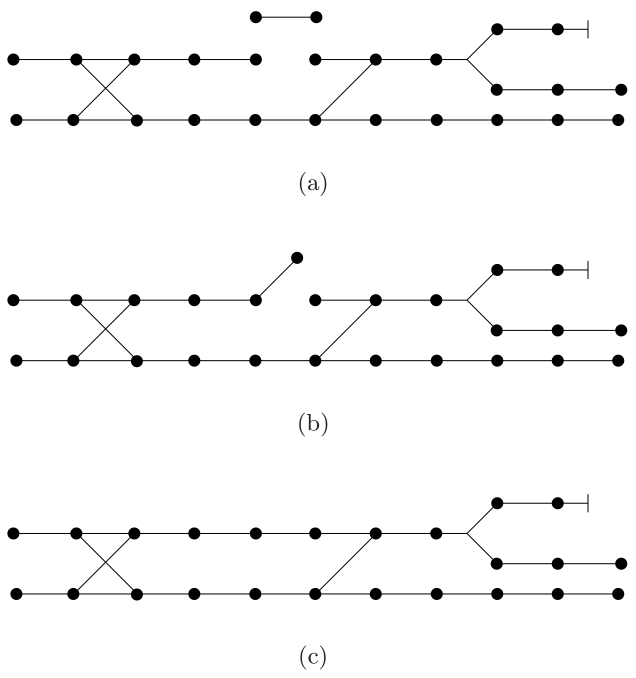


Figure 3.4: Creating a loop specification.

connect two different specification graphs, as for instance by connecting the two graphs of Figure 3.4(a) into the graph of Figure 3.4(b), but to be able to get from the situation of Figure 3.4(b) to the situation of Figure 3.4(c) the specification graph must be connected to itself.

Lemma 2 *Composition of joinable specifications graphs forms an partial* *abelian monoid.*

Proof: For $\sqcap_b$ to be an partial abelian monoid, it must fulfill the following four properties, closure, identity, associativity and commutativity

Closure: For all joinable specifications graphs G_1 and G_2, $\mathrm{G}_1 \sqcap_b \mathrm{G}_2$ is a specification graph.

Identity element: $\mathrm{G} \sqcap_b \mathrm{G}_\emptyset = \mathrm{G} = \mathrm{G}_\emptyset \sqcap_b \mathrm{G}$

Associativity: $(\mathrm{G}_1 \sqcap_{b_1} \mathrm{G}_2) \sqcap_{b_2} \mathrm{G}_3 = \mathrm{G}_1 \sqcap_{b_1} (\mathrm{G}_2 \sqcap_{b_2} \mathrm{G}_3)$

Commutativity: $\mathrm{G}_1 \sqcap_b \mathrm{G}_2 = \mathrm{G}_2 \sqcap_b \mathrm{G}_1$

Closure:

If $\mathrm{G}_1 = \langle \mathrm{N}_1, \mathrm{N}_1^{\mathrm{Int}}, \mathrm{L}_1 \rangle$ and $\mathrm{G}_2 = \langle \mathrm{N}_2, \mathrm{N}_2^{\mathrm{Int}}, \mathrm{L}_2 \rangle$ are two joinable specification graphs, there exists interface nodes $n_1 \in \mathrm{N}_1$ and $n_2 \in \mathrm{N}_2$ such that n_1 and n_2 are joinable. Then composition of G_1 and G_2 by the binding $b = [n_1, n_2]$ is $sub(n_2, n_1 \textcircled{n} n_2, (sub(n_1, n_1 \textcircled{n} n_2, \mathrm{G}_1 \sqcup \mathrm{G}_1)))$, which is a graph since the flat union of two graphs always is a graph, and the result of replacing a node in a graph also is a graph.

Identity element:

We need to prove that there exists an identity element $\mathrm{G}_\emptyset$, and that
$\mathrm{G}_1 \sqcap_\epsilon \mathrm{G}_\emptyset = \mathrm{G}_1 = \mathrm{G}_\emptyset \sqcap_\epsilon \mathrm{G}_1$

$$\begin{aligned}
\mathrm{G}_1 \sqcap_\epsilon \mathrm{G}_\emptyset &\overset{1}{=} sub(\epsilon, \epsilon, sub(\epsilon, \epsilon, \mathrm{G}_1 \sqcup \mathrm{G}_\emptyset)) \\
&\overset{2}{=} sub(\epsilon, \epsilon, sub(\epsilon, \epsilon, \mathrm{G}_\emptyset \sqcup \mathrm{G}_1)) \\
&\overset{3}{=} \mathrm{G}_\emptyset \sqcap_\epsilon \mathrm{G}_1
\end{aligned}$$

$$\begin{aligned}
\mathrm{G}_1 \sqcap_\epsilon \mathrm{G}_\emptyset &\overset{1}{=} sub(\epsilon, \epsilon, sub(\epsilon, \epsilon, \mathrm{G}_1 \sqcup \mathrm{G}_\emptyset)) \\
&\overset{2}{=} sub(\epsilon, \epsilon, sub(\epsilon, \epsilon, \mathrm{G}_1)) \\
&\overset{3}{=} \mathrm{G}_1
\end{aligned}$$

Associativity:

Let the specifications graphs $\mathrm{G}_1 = \langle \mathrm{N}_1, \mathrm{N}_1^{\mathrm{Int}}, \mathrm{L}_1 \rangle$ and $\mathrm{G}_2 = \langle \mathrm{N}_2, \mathrm{N}_2^{\mathrm{Int}}, \mathrm{L}_2 \rangle$ be joinable with the binding $b_1 = [n_1, n_2]$, where $n_1 \in \mathrm{N}_1$ and $n_2 \in \mathrm{N}_2$, and G_2 and $\mathrm{G}_3 = \langle \mathrm{N}_3, \mathrm{N}_3^{\mathrm{Int}}, \mathrm{L}_3 \rangle$ be joinable with the binding $b_2 = [n_{2'}, n_3]$, where $n_{2'} \in \mathrm{N}_2$ and $n_3 \in \mathrm{N}_3$.

We must add the restriction that $n_1 \neq n_2 \neq n_{2'} \neq n_3$ and that $\mathrm{G}_1 \neq \mathrm{G}_2 \neq \mathrm{G}_3$. The restrictions are necessary because the term $(\mathrm{G}_1 \sqcap_{b_1} \mathrm{G}_2) \sqcap_{b_2} \mathrm{G}_3$ is meaningless if for instance $\mathrm{G}_1 = \mathrm{G}_3$. G_3 will then not exist after G_1 and G_2 are joined, and can therefore not be joined with the composite graph of G_1 and G_2. The same is applicable for the nodes.

We must then show that $(\mathrm{G}_1 \sqcap_{b_1} \mathrm{G}_2) \sqcap_{b_2} \mathrm{G}_3 = \mathrm{G}_1 \sqcap_{b_1} (\mathrm{G}_2 \sqcap_{b_2} \mathrm{G}_3)$

$$\begin{aligned}
(\mathrm{G}_1 \sqcap_{b_1} \mathrm{G}_2) \sqcap_{b_2} \mathrm{G}_3 &\overset{1}{=} sub(n_2, n_1 \textcircled{n} n_2, sub(n_1, n_1 \textcircled{n} n_2, \mathrm{G}_1 \sqcup \mathrm{G}_2)) \sqcap_{b_2} \mathrm{G}_3 \\
&\overset{2}{=} sub(n_3, n_{2'} \textcircled{n} n_3, sub(n_{2'}, n_{2'} \textcircled{n} n_3, \\
&\quad (sub(n_2, n_1 \textcircled{n} n_2, sub(n_1, n_1 \textcircled{n} n_2, \mathrm{G}_1 \sqcup \mathrm{G}_2))) \\
&\quad \sqcup \mathrm{G}_3)) \\
&\overset{3}{=} sub(n_3, n_{2'} \textcircled{n} n_3, sub(n_{2'}, n_{2'} \textcircled{n} n_3, \\
&\quad sub(n_2, n_1 \textcircled{n} n_2, sub(n_1, n_1 \textcircled{n} n_2, \mathrm{G}_1 \sqcup \mathrm{G}_2 \sqcup \mathrm{G}_3)))) \\
&\overset{4}{=} sub(n_2, n_1 \textcircled{n} n_2, sub(n_1 n_1 \textcircled{n} n_2, \\
&\quad sub(n_3, n_{2'} \textcircled{n} n_3, sub(n_{2'}, n_{2'} \textcircled{n} n_3, \\
&\quad \mathrm{G}_1 \sqcup \mathrm{G}_2 \sqcup \mathrm{G}_3)))) \\
&\overset{5}{=} sub(n_2, n_1 \textcircled{n} n_2, sub(n_1 n_1 \textcircled{n} n_2, \mathrm{G}_1 \sqcup \\
&\quad (sub(n_3, n_{2'} \textcircled{n} n_3, sub(n_{2'}, n_{2'} \textcircled{n} n_3, \mathrm{G}_2 \sqcup \mathrm{G}_3))))) \\
&\overset{6}{=} \mathrm{G}_1 \sqcap_{b_1} \\
&\quad sub(n_3, n_{2'} \textcircled{n} n_3, sub(n_{2'}, n_{2'} \textcircled{n} n_3, \mathrm{G}_2 \sqcup \mathrm{G}_3)) \\
&\overset{7}{=} \mathrm{G}_1 \sqcap_{b_1} (\mathrm{G}_2 \sqcap_{b_2} \mathrm{G}_3)
\end{aligned}$$

Equation 1 follows by Definition 26, and equation 2 is filling in for $\sqcap_{b_2}$. Equation 3 holds because $n_{2'}$ and n_3 are not the same nodes as n_1 and n_2, and because $\sqcup$ is associative. Equation 4 is valid because of the associativity of *sub*. Equation 5 holds because neither $n_{2'}$ or n_3 is present in G_1. Equation 6 and 7 follows from the definition of $\sqcap_b$.

Commutativity:

Let the specification graphs $\mathrm{G}_1 = \langle \mathrm{N}_1, \mathrm{N}_1^{\mathrm{Int}}, \mathrm{L}_1 \rangle$ and $\mathrm{G}_2 = \langle \mathrm{N}_2, \mathrm{N}_2^{\mathrm{Int}}, \mathrm{L}_2 \rangle$ be joinable with the binding $b = [n_1, n_2]$, where $n_1 \in \mathrm{N}_1$ and $n_2 \in \mathrm{N}_2$. Then:

$$\begin{aligned} \mathrm{G}_1 \sqcap_b \mathrm{G}_2 &\overset{1}{=} sub(n_2, n_1 ⓝ n_2, sub(n_1, n_1 ⓝ n_2, \mathrm{G}_1 \sqcup \mathrm{G}_2)) \\ &\overset{2}{=} sub(n_2, n_1 ⓝ n_2, sub(n_1, n_1 ⓝ n_2, \mathrm{G}_2 \sqcup \mathrm{G}_1)) \\ &\overset{3}{=} \mathrm{G}_2 \sqcap_b \mathrm{G}_1 \end{aligned}$$

This is valid because of the commutativity of the union of graphs. ♣

Definition 27 *The copy function $\mathscr{C}$ makes a copy of a graph. All the nodes of the new graph gets new ids, but otherwise it is equal to the original.*

Definition 28 *The template function $\mathscr{T}$ returns the template of a graph if the graph is a copy of a template. It is undefined for template graphs and composed graphs.*

The specification graphs may now be constructed from the empty specification by joining the specification graph with a copy of one of the template graphs.

3.3 The specification language used to model the Oslo subway system

The language used to model the subway system of Oslo consists of:

- 2 directions
- 26 types
- 1 combination rule
- 0 structural exceptions
- 9 template graphs

All of these are presented in this section. This language is designed to model the subway system of Oslo, and may be reduced or expanded to fit other railroad systems.

3.3.1 Directions and types

The language used to represent Oslo subway system has two directions. These are the positive direction, denoted 1, and the negative direction, denoted −1.

$$\mathrm{D} = \{1, -1\}$$

The set of types (T) is rather large. This is because all the interface nodes of all the template components of the language has it's own type. This is done to prevent putting restrictions to the structural exceptions.

$$\begin{aligned}\mathrm{T} = \{ \quad & Tr1, Tr2, \\ & En, \\ & Rc1, Rc2, Rc3, Rc4, \\ & Tu1, Tu2, Tu3, \\ & Rs1, Rs2, Rs3, Rs4, \\ & Ls1, Ls2, Ls3, Ls4, \\ & Sc1, Sc1, Sc3, Sc4, \\ & Nn1, Nn2, \\ & Rr1, Rr2 \quad \}\end{aligned}$$

3.3.2 Rules and exceptions

The specification language contains only one combination rule. This rule expresses that in order to connect two interface nodes, they must have opposite directions.

$$\mathrm{C} = \{\langle 1, -1 \rangle\}$$

When modeling an existing railway system, structural exceptions is redundant. The engineers that designed the system have followed some rules, but adding restrictions to the specification language could result in that the language is no longer suitable to model the given railway. The language used to model the subways of Oslo therefore contains no structural exceptions.

$$\mathrm{E} = \emptyset$$

3.3.3 Template components

The tracks of the Oslo subway system is built up by numerous copies of seven different railroad components. These are the track segment, the end segment, the rigid crossing, the turnout, the left and the right slip, and finally the scissors. In the specification language, each of these seven railroad components are represented by a template graph. The language also have two special components called the negative and the positive converter.

Track segment

The track segment is the main building block of railway networks. This is a single line, straight or curved. It can be driven in both directions, but it is impossible to pass another train on a track segment. The leftmost part of

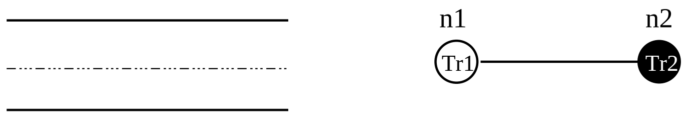


Figure 3.5: The track segment

Figure 3.5 shows the track segment as seen from above. It consists of two parallel rails with no branches. A graphical representation of graph modeling the track segment is shown in the right hand side of Figure 3.5. This graph has two nodes, and one line. The color of the nodes indicates the direction of the node, a white node has the negative direction, and a black node has the positive direction. The name of the node is printed above it. The text inside the node gives the type. The text representation of the track segment of Figure 3.5 is:

$$\langle \{\langle n1, Tr1, -1\rangle, \langle n2, Tr2, 1\rangle\}, \{\emptyset\}, \{\langle n1, n2\rangle\}, \rangle$$

End segment

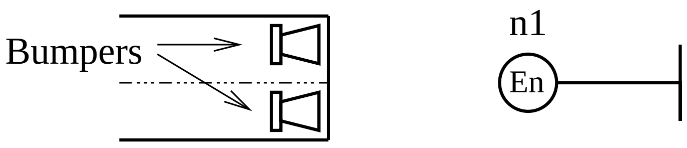


Figure 3.6: The end segment

The end segments is used at the end of the tracks. Is is equipped with bumpers to prevent derailing. The text representation of the end segment of Figure 3.6 is:

$$\langle\{\langle n1, En, -1\rangle\}, \{\emptyset\}, \{\langle n1, n1\rangle\}\rangle$$

The end segment consists of one interface node ($n1$) and a line from $n1$ to $n1$.

Rigid crossing

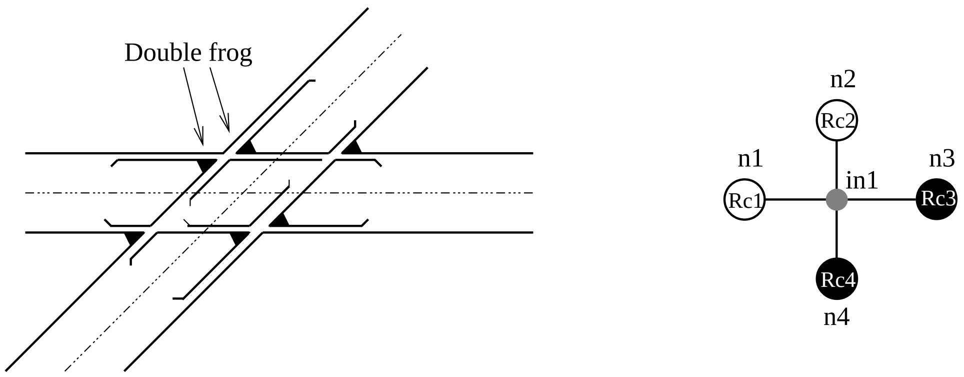


Figure 3.7: The rigid crossing

The rigid crossing is a crossing of two tracks without movable points. This means that a train that enters a rigid crossing has only one possible exit point. The text representation of the rigid crossing of Figure 3.7 is:

$$\langle\{\langle n1, Rc1, -1\rangle, \langle n2, Rc2, -1\rangle, \langle n3, Rc3, 1\rangle, \langle n4, Rc4, 1\rangle\},$$
$$\{in1\}, \{\langle n1, in1\rangle, \langle n2, in1\rangle, \langle n3, in1\rangle, \langle n4, in1\rangle\}\rangle$$

This component consists of four interface nodes ($n1, n2, n3$ and $n4$) and one internal node ($in1$). There are lines from all the interface nodes to the internal node.

Turnout

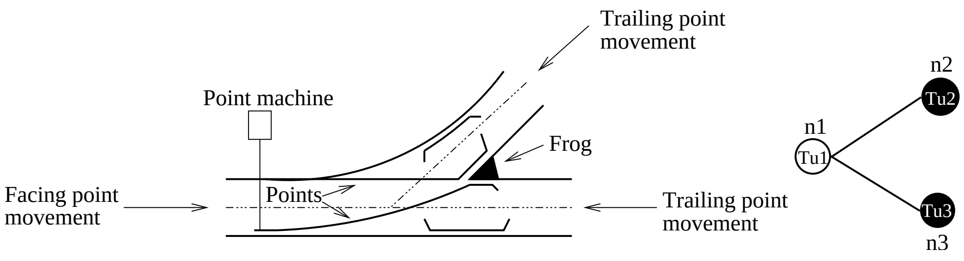


Figure 3.8: The turnout

The turnout is an assembly of rails, movable points and a frog. When a train enters a turnout from the stem end, it can exit the turnout from any of the two branching ends. Which of the branching ends it will exit from depends on the position of the points. The points can either be operated manually or by a mechanical device called a *point machine*.

The text representation of the turnout of Figure 3.8 is:

$$\langle\{\langle n1, Tu1, -1\rangle, \langle n2, Tu2, 1\rangle, \langle n3, Tu3, 1\rangle\}, \{\emptyset\}, \{\langle n1, n2\rangle, \langle n1, n3\rangle\}\rangle$$

The turnout has three interface nodes ($n1, n2$ and $n3$), and lines from $n1$ to both $n2$ and $n3$.

Right slip

The right slip is an assembly of two turnouts. The right slip is usually used to allow a train to pass from the left track to the right track in a system with two parallel tracks. The text representation of the right slip of Figure 3.9 is:

$$\langle\{\langle n1, Rs1, -1\rangle, \langle n2, Rs2, 1\rangle, \langle n3, Rs3, -1\rangle, \langle n4, Rs4, 1\rangle\}, \\ \{\emptyset\}, \{\langle n1, n2\rangle, \langle n1, n4\rangle, \langle n3, n4\rangle\}\rangle$$

The right slip has four interface nodes ($n1, n2, n3$ and $n4$). There are lines from $n1$ to $n2$ and $n4$, and a line from $n3$ to $n4$.

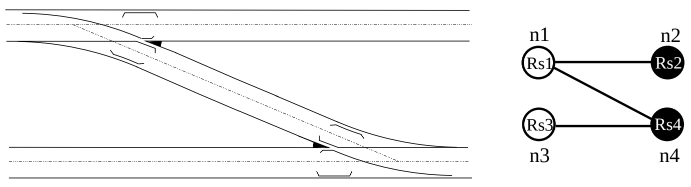


Figure 3.9: The right slip

Left slip

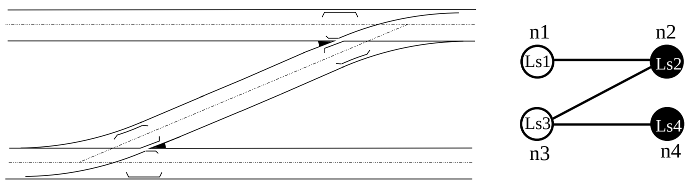


Figure 3.10: The left slip

The left slip is the mirror image of the right slip, it's purpose is also the inverted of the right slip's. The text representation of the left slip of Figure 3.10 is:

$$\langle\{\langle n1, Ls1, -1\rangle, \langle n2, Ls2, 1\rangle, \langle n3, Ls3, -1\rangle, \langle n4, Ls4, 1\rangle\},\\ \{\emptyset\}, \{\langle n1, n2\rangle, \langle n2, n3\rangle, \langle n3, n4\rangle\}\rangle$$

The left slip has four interface nodes ($n1, n2, n3$ and $n4$). There are lines from $n1$ to $n2$, from $n2$ to $n3$, and a from $n3$ to $n4$.

Scissors

The scissors is a compound component which is built up by four turnouts and a rigid crossing. A scissors component will allow trains to move from one track to the other regardless of the direction of the train. The text representation of the scissors of Figure 3.11 is:

$$\langle \quad \{\langle n1, Sc1, -1\rangle, \langle n2, Sc2, 1\rangle, \langle n3, Sc3, -1\rangle, \langle n4, Sc4, 1\rangle\},\\ \{in1\},\\ \{\langle n1, n2\rangle, \langle n3, n4\rangle, \langle n1, in1\rangle, \langle n2, in1\rangle, \langle n3, in1\rangle, \langle n4, in1\rangle\} \quad \rangle$$

The scissors component has four interface nodes ($n1, n2, n3$ and $n4$) and one internal node ($in1$). There are lines from $n1$ to $n2$ and from $n3$ to $n4$. As in the rigid crossing there are also lines from all the interface nodes to the internal node. A typical use of scissors is to enable re-routing in the case of

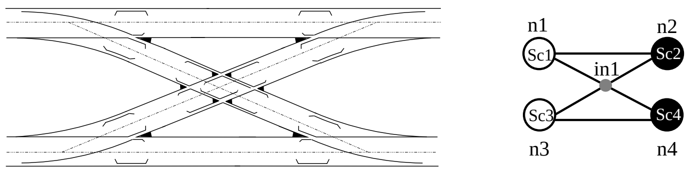
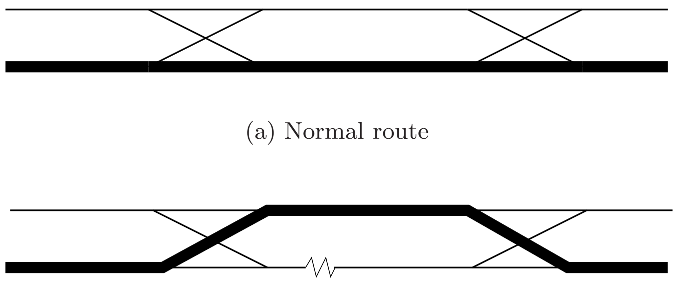


Figure 3.11: The scissors

a blocked line. Figure 3.12(a) shows the normal route of a train through a part of a system. The route is indicated by the thick line. If for some reason



(a) Normal route

(b) Rerouted

Figure 3.12: Re-routing by using scissors

the line between the two scissors is blocked, represented by the broken line of Figure 3.12(b), the train may be rerouted as the thick line shows.

Direction converters

Direction converters are not actually railroad components, but structural components used to redefine the direction. If traveling to the left is defined as traveling in the positive direction on one side of a converter it will be defined as traveling in the negative direction on the other side of the converter. There are two direction converters in the language. The first one is the negative converter, in which both directions are defined to be in the negative direction.

$$\langle\{\langle n1, Nn1, -1\rangle, \langle n2, Nn2, -1\rangle\}, \{\emptyset\}, \{\langle n1, n2\rangle\}\rangle$$

The other is the positive converter. Here both directions are defined as positive.

$$\langle\{\langle n1, Pp1, 1\rangle, \langle n1, Pp2, 1\rangle\}, \{\emptyset\}, \{\langle n1, n2\rangle\}\rangle$$

Figure 3.13 shows a graphical representation of the direction converters, the

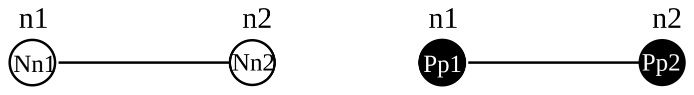


Figure 3.13: Direction converters

negative converter to the left, and the positive converter to the right.

The direction converters can be used to solve the problems with direction traps. The direction trap that arises in Figure 3.2 can be solved by inserting a negative direction converter between the two track segments. This is shown in Figure 3.14. If a train enters the node `n2○n5` from the node `n1`, it is

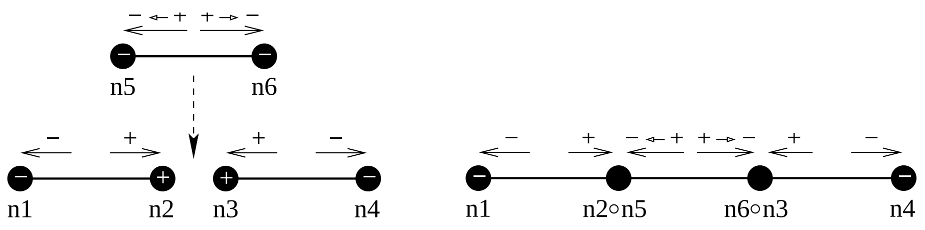
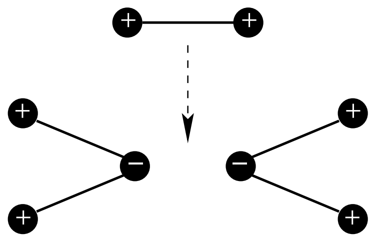


Figure 3.14: Solution to the direction trap

traveling in the positive direction. It may then travel further towards the node `n6○n3`, as it does the direction of the train is redefined to the negative direction. When the train arrives at the node `n6○n3` it is traveling in the negative direction, and can then proceed towards the node `n4`, as intended. Note that the two nodes `n2` and `n3` can not be joined because their directions does not form a pair in the combination rules.


(a) Solution with converters

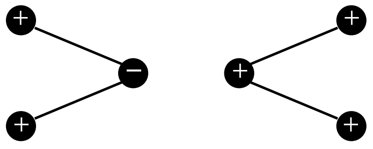

(b) Solution without converters

Figure 3.15: Different solutions to the direction trap

One could argue that to solve the direction trap situation in Figure 3.14, one could simply turn one of the track segments around. Then one could connect the nodes `n2` and `n4`. This would be a solution if all components where symmetrical. This however is not the case. A typical situation where a direction trap may occur, is when two turnouts shall be connected in their stem end. There are two solutions to this problem. Figure 3.15(a) shows the approach we have chosen. Here the problem is solved by inserting a direction converter between the two turnouts. Another approach is shown in Figure 3.15(b). Here a new version of the turnout component is introduced. Note that the turnout on the righthand side of the figure have different directions than the others. This makes it possible for the two turnouts to connect by joining their stem ends. All possible direction traps may be solved this way. This would however result in a large increase in the number of template components. This is why we have chosen the direction converters.

Chapter 4

Railroad nets

4.1 Introduction

A *railroad net* is an extension of a colored Petri net as discussed in Section 2.2. In a railroad net, places may be compounded into groups, called *interfaces*, in order to connect several places at once when two nets are connected to each other. This can be useful when modeling railroads. In for instance, Section 5.4 railroad components with signals in a separate path parallel to the tracks are introduced. When connecting two such nets, we must connect both the tracks and the signal path. The same end of both the tracks and the signals path are therefore combined in the same interface. For instance, the the `Left` place and the `LeftSignal` place of Figure 5.6 are combined in the same interface.

Definition 29 *An interface* i *is a set of places such that* $\pi_2(p_1) \neq \pi_2(p_2)$ *for all* $p_1, p_2 \in \mathrm{i}$.

The empty interface $I_\emptyset$ is a set that contains only the empty place ($P_\emptyset$). All the places in an interface must have different colors. The reason for this is thoroughly described in Section 4.3. Places can not be in more than one interface. Interfaces are drawn as dotted rectangles with rounded corners. For an example see the railroad net in Figure 4.1.

Definition 30 *A railroad net* R *is a pair* $\langle \mathrm{CPN}, \mathrm{I} \rangle$*, where*

1. CPN $= \langle P, T, A \rangle$ *is a colored Petri net as defined in Definition 14*

2. I $= \{\mathrm{i}_1, \ldots, \mathrm{i}_n\}$ *is a set of interfaces, where* $\forall p \in \mathrm{i}_j, p \in P$ *for all* j *such that* $0 \leq j \leq n$*, and* $\mathrm{i}_j \cap \mathrm{i}_{j'} = \emptyset$ *for all* $0 \leq j \leq n$ *and* $0 \leq j' \leq n$ *such that* $j \neq j'$*.*

Condition number two in the definition above simply states that in order for a place to be in an interface, it must be a place of the colored Petri net, and that a place can not be in more than one interface. The empty railroad net consists of the empty colored Petri net and the empty interface ($N_\emptyset = \langle \mathrm{CPN}_\emptyset, I_\emptyset \rangle$).

A *library* ($\mathscr{L}$) is a set of railroad nets and a declaration file. The railroad nets of a library usually have a one to one correspondence to the template graphs of a specification language. The relationship between the two are described in Chapter 6.1. All railroad nets that are members of a library is said to be *atomic*. The declaration file contains declarations of colors and functions used in the railroad nets.

4.2 Atomic railroad nets

The nine railroad nets of this section forms a library referred to as *the basic components*. They have a one to one correspondence to the template components described in Section 3.3.

All the railroad nets of the basic components library can be divided into two parts. The track part, and the controlling part. The track part consists of places that represents points along the track, and transitions that takes care of the movement of trains. Places of the track part are referred to as *track places*, and the transitions are called *move transitions*. Trains are represented as tokens. A train token has a name and a direction. The name is only a text used to recognize the train, but the direction indicates which way the train is heading. There are three directions the positive, the negative and zero. Train tokens may only occupy track places. The controlling parts of the system models point machines, and other structures made to control the movement.

4.2.1 Track segment

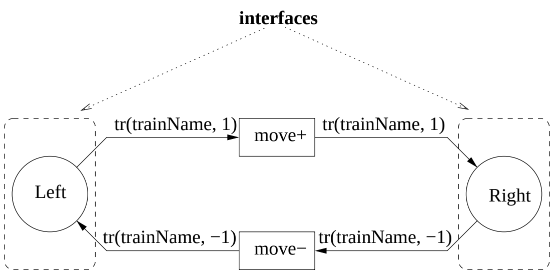


Figure 4.1: The track segment

Figure 4.1 shows the track segment as a railroad net. The two places `Left` and `Right` are the two end points of the segment. These places are track places because they represent points of the track. The `move+` transition moves a train token, that travels in the positive direction, from the `Left` place to the `Right` place. The `move-` transition moves a train token, that travels in the negative direction, from the `Right` place to the `Left` place. Note that if the directions were omitted the trains could run back and forth between the `Left` place and the `Right` place. The railroad net contains two interfaces, one that contains the `Left` place, and another that contains the `Right` place. This is the case for all the basic atomic railroad nets. All track places belongs to different interfaces.

4.2.2 End segment

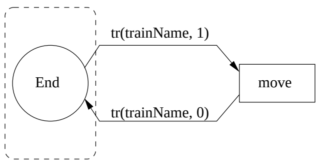


Figure 4.2: The end segment

Figure 4.2 shows a railroad net representation of the end segment. It consists of one track place, the `End` place, and one move transition. This transition takes a train that is moving in the positive direction from the `End` place, and gives it the direction zero. This means that the train has stopped, and is no longer traveling in any direction. In other words trains stop when they reach an end segment. In some situations one could want the trains to turn when they reach an end segment. This can be done by letting the train move in the negative direction after the firing of the `move` transition.

4.2.3 Rigid crossing

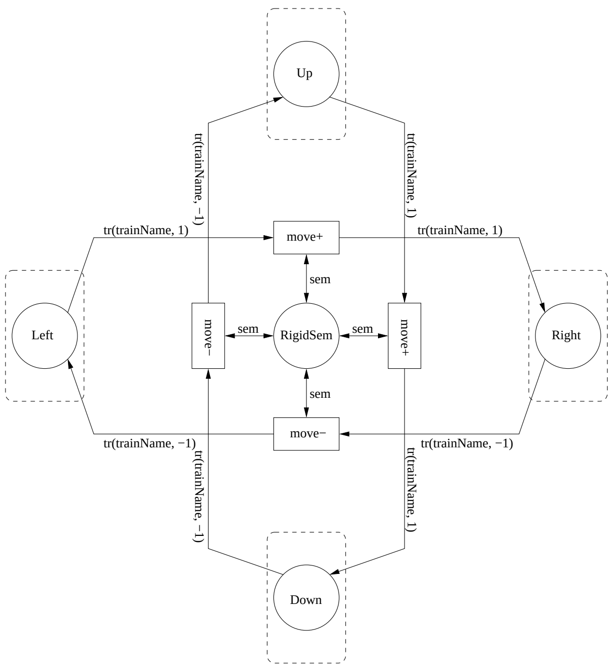


Figure 4.3: The rigid crossing

The rigid crossing of Figure 4.3 is a composition of two track segments. The place `RigidSem` is a semaphore place, and is a part of the control system. This place should always contain one, and only one semaphore token. The purpose of this place is to keep the railroad net connected, and to prevent two trains one on each track to drive through the crossing simultaneously.

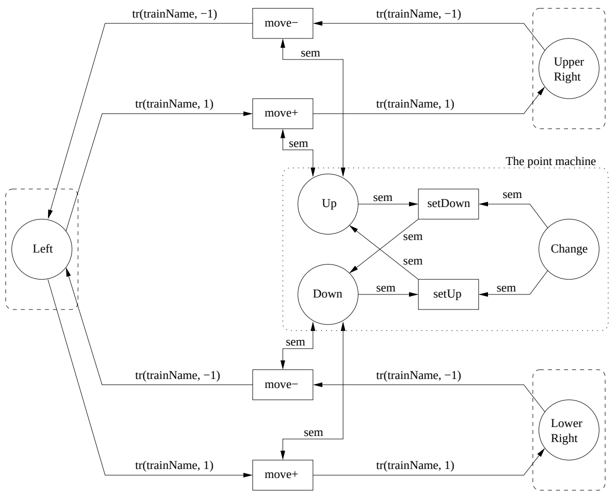


Figure 4.4: The turnout

4.2.4 Turnout

Figure 4.4 shows a railroad net representation of a turnout. This is basically two track segments, such as the one in Figure 4.1, joined in their `Left` places, and a point machine. The point machine is a part of the control system, and works as follows. The places `Up` and `Down` gives us the state of the turnout. One of these should always contain a semaphore token. In order for a train to travel from `Left` to `UpperRight` or from `UpperRight` to `Left` is has to be a semaphore token in the `Up` place. In the same way it has to be a semaphore token in the `Down` place for a train to be able to run from `Left` to `LowerRight` or vice versa.

To change the state of the turnout, simply add a semaphore token to the `Change` place. This will enable one of the two transitions `setUp` or `setDown`. The firing of one of these transitions will result in a change of the state of the turnout.

There is a possibility of a dead-lock in this component. If train that travels in the negative direction is in the `UpperRight` place, and there are no semaphore token in the `Up` place, the train can not move. One must then manually add a semaphore token to the `Change` place in order to get the train moving again. An automatic solution to this is presented in Section 5.2. A similar dead-lock situation arises when a train that travels in the negative direction is in the `LowerLeft`, and there are no semaphore token in the `Down` place.

4.2.5 Direction converters

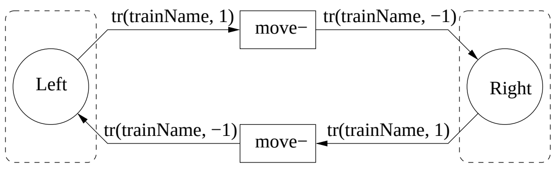


Figure 4.5: The negative converter

The direction converters always invert the direction of trains traveling them. The direction converter in Figure 4.5 is called the negative converter. This is because when it moves a train it always ends up traveling in the negative direction. The positive converter is the opposite of the negative converter. Trains driving trough a positive converter always ends up in the positive direction. The positive converter is displayed in Figure 4.6. The only difference between the converters and the track segment are the arc expressions.

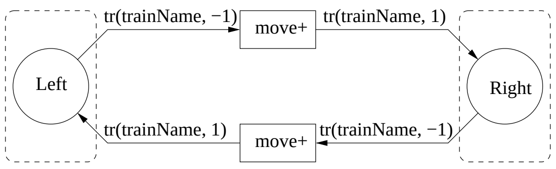


Figure 4.6: The positive converter

4.2.6 Singles

The singles are compounded by two turnouts. The Figure 4.7 shows the single right. Trains in the `UpperLeft` or the `LowerRight` may either travel straight forward or turn right dependent of the states of the point machines. The single left is a mirror image of the single right.

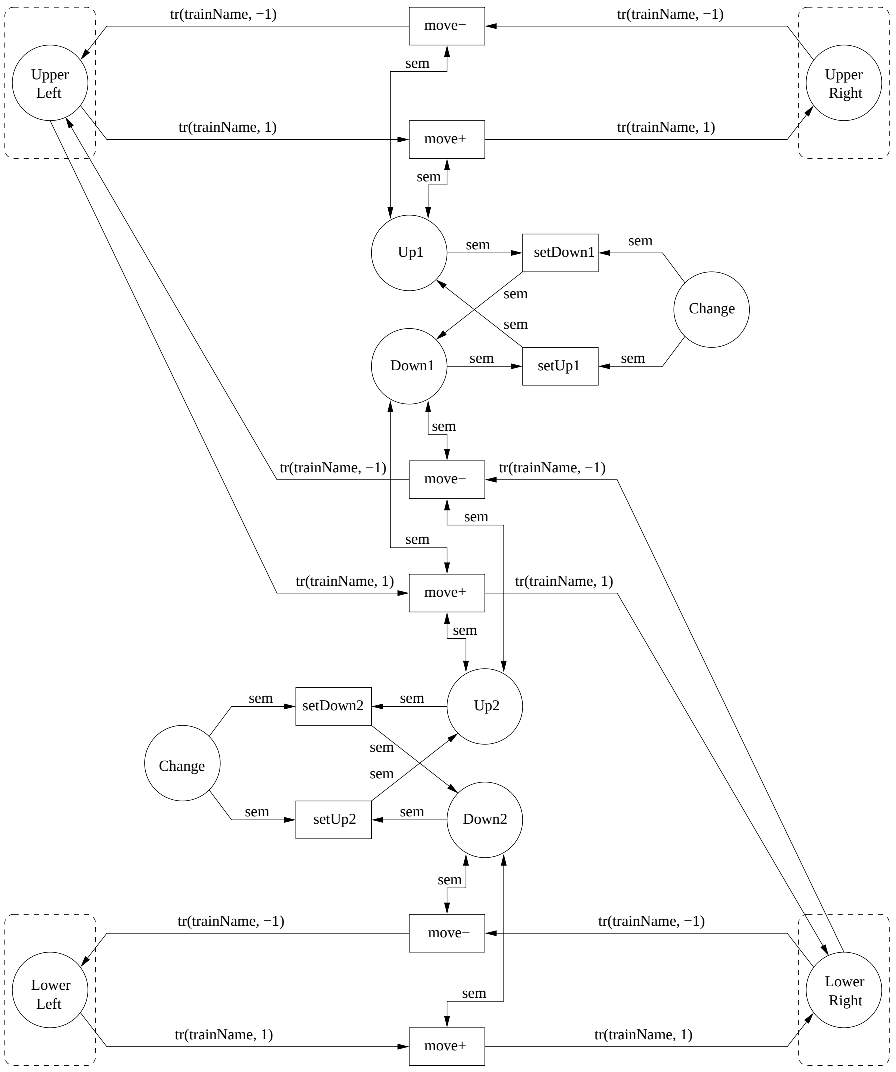


Figure 4.7: The single right

The singles inherits the dead-lock situations from the turnouts.

4.2.7 Scissors

Figure 4.8 shows the railroad net representation of a scissor component. This is built up by four turnouts and a rigid crossing. Trains that enter this component may either travel straight forward or diagonally across the component. Again this is dependent on the states of the turnouts.

4.3 Composition of railroad nets

In order to connect copies of these atomic railroad nets into larger nets, two places have to be merged into a union place, and all the arcs with an endpoint in one of the merged places must be redirected to the new place. The first thing we need to define is which places are joinable. Two places are joinable if they do have the same color, and if they are marked, their tokens must be indistinguishable. This is formally defined in Definition 31

Definition 31 *Two places p_1 of a net N_1 and p_2 of a net N_2 are joinable if:*

1. $\pi_1(p_1) \neq \pi_1(p_2)$
2. $\pi_2(p_1) = \pi_2(p_2)$

if the net N_1 is marked with the marking M_1, and N_2 is marked with M_2 then we also must require that:

3. $\pi_2(\mathbb{m}(p_1, M_1)) = \pi_2(\mathbb{m}(p_2, M_2))$

Let p_1 and p_2 be two joinable places. If they are joined with the place join operator ⓟ, they will merge into a new union place. The id of the union place will be the ids of the original places, concatenated by the unordered concatenation operator $\circ$, as defined in Definition 12. The colors of p_1 and p_2 are the same, by the definition of joinable places. The union place will also get this color. If the the places p_1 and p_2 contains any tokens, these

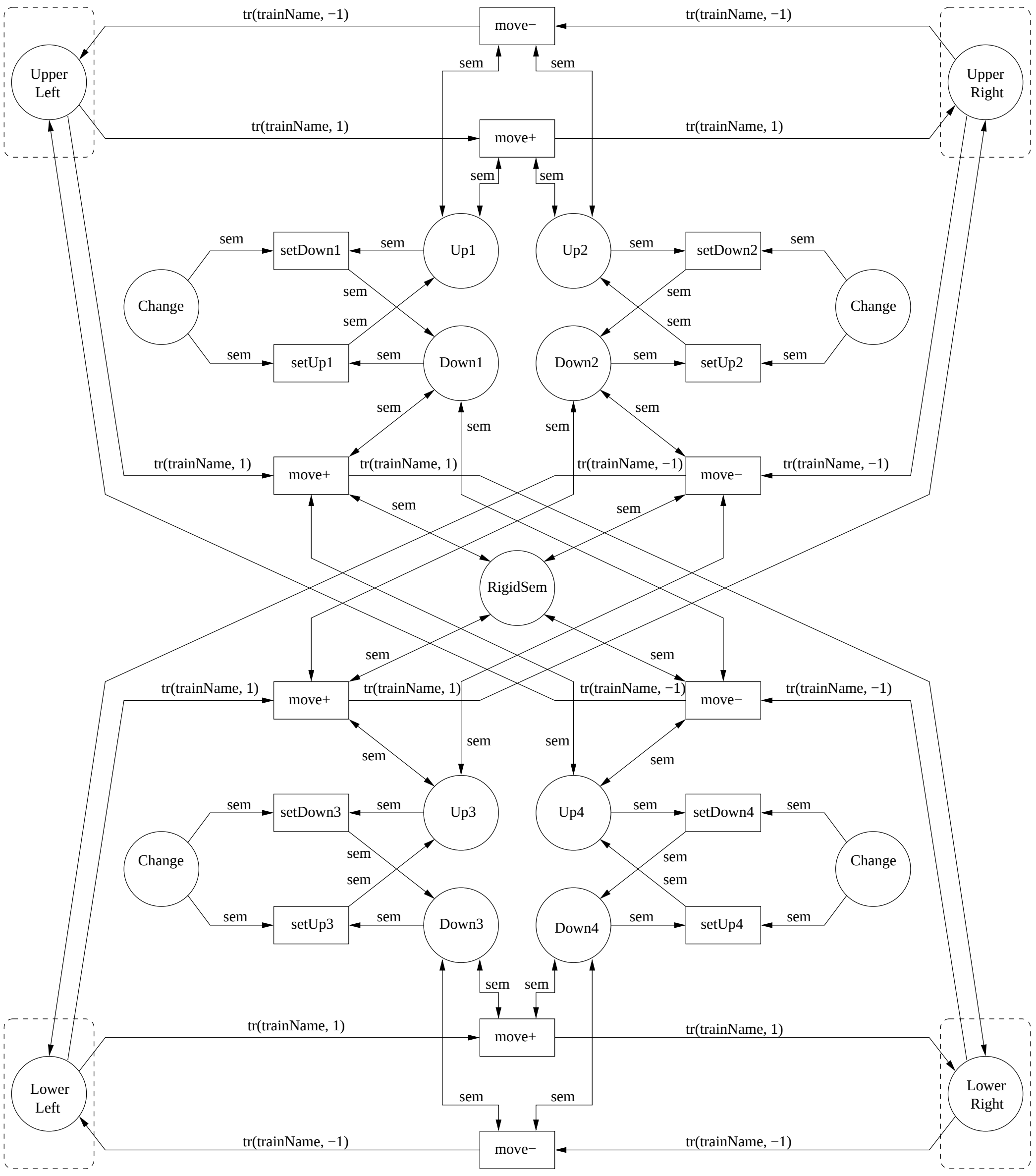


Figure 4.8: The scissors

tokens are indistinguishable. The new union place gets the tokens from one of the original places.

Definition 32 *Let $p_1 = \langle id_1, color_1 \rangle$ and $p_2 = \langle id_2, color_2 \rangle$ be two joinable places. Then $p_1 \textcircled{p} p_2 = \langle id_1 \circ id_2, color_1 \rangle$. If p_1 is a place of a net marked with M_1 and p_2 is a place of a net marked with M_2 then $\mathbb{m}(p_1 \textcircled{p} p_2, M_1) = \langle p_1 \textcircled{p} p_2, \pi_2(\mathbb{m}(p_1, M_1)) \rangle$ and $\mathbb{m}(p_1 \textcircled{p} p_2, M_2) = \langle p_1 \textcircled{p} p_2, \pi_2(\mathbb{m}(p_1, M_1)) \rangle$.*

Two interfaces are joinable if they contain the same number of places, and all places in one interface can be joined with exactly one place in the other interface.

Definition 33 *Two interfaces i_1 and i_2 are joinable if*

1. $\mid \mathrm{i}_1 \mid = \mid \mathrm{i}_2 \mid$
2. $\forall p \in \mathrm{i}_1 \exists p' \in \mathrm{i}_2$ *such that p and p' are joinable.*
3. $\forall p \in \mathrm{i}_2 \exists p' \in \mathrm{i}_1$ *such that p and p' are joinable.*

Note that because the interfaces have the same number of nodes, and all places of an interface have different colors, all places of two joinable interfaces are joinable with one and only one place. The operator $\boxed{\mathbb{p}}$ takes two joinable interfaces, and returns a set of pairs of joinable places.

Definition 34 *Let i_1 and i_2 be two joinable interfaces. Then $i_1 \boxed{\mathbb{p}}\, i_2$ is defined by:*

1. $(\{p\} \cup P) \boxed{\mathbb{p}}\, P' = (p \boxed{\mathbb{p}}\, P') \cup ((P \backslash \{p\}) \boxed{\mathbb{p}}\, P')$
2. $p \boxed{\mathbb{p}}\, (\{p'\} \cup P') = p \boxed{\mathbb{p}}\, (P' \backslash \{p'\})$ *if* $\pi_2(p) \neq \pi_2(p')$
3. $p \boxed{\mathbb{p}}\, (\{p'\} \cup P') = \{\langle p, p' \rangle\}$ *if* $\pi_2(p) = \pi_2(p')$
4. $p \boxed{\mathbb{p}}\, \{P_\emptyset\} = \{\langle p, P_\emptyset \rangle\}$
5. $\{P_\emptyset\} \boxed{\mathbb{p}}\, P = P \boxed{\mathbb{p}}\, \{P_\emptyset\}$,

where p and p' are places and P and P' are sets of places.

Let i_1 be an interface of net N_1 and i_2 be an interface of net N_2. If i_1 and i_2 are joinable we say that N_1 and N_2 are joinable with the binding $b = [i_1, i_2]$. We also need a replacement function *sub* which takes an old place, a new place and a colored Petri net as arguments. It will replace the old place by the new place. It will also redirect all arcs with an end-point in the old place.

Definition 35 *The replacement function sub is defined by the following equations:*

1. $sub(p_{old}, p_{new}, \langle P, T, A\rangle) = \langle sub(p_{old}, p_{new}, P), T, sub(p_{old}, p_{new}, A)\rangle$
2. $sub(p_{old}, p_{new}, \{p_{old}\} \cup P) = \{p_{new}\} \cup (P \backslash \{p_{old}\})$
3. $sub(p_{old}, p_{new}, \{p\} \cup P) = \{p\} \cup sub(p_{old}, p_{new}, P \backslash \{p\})$ *if* $p_{old} \neq p$
4. $sub(p_{old}, p_{new}, \{\langle t, p_{old}, e\rangle\} \cup A) =$
   $\{\langle t, p_{new}, e\rangle\} \cup \; sub(p_{old}, p_{new}, A \backslash \{\langle t, p_{old}, e\rangle\})$
5. $sub(p_{old}, p_{new}, \{\langle p_{old}, t, e\rangle\} \cup A) =$
   $\{\langle p_{new}, t, e\rangle\} \cup \; sub(p_{old}, p_{new}, A \backslash \{\langle t, p_{old}, e\rangle\})$
6. $sub(p_{old}, p_{new}, \{\langle n, n', e\rangle\} \cup A) =$
   $\{\langle n, n', e\rangle\} \cup \; sub(p_{old}, p_{new}, A \backslash \{\langle n, n', e\rangle\})$ *if* $p_{old} \neq n \wedge p_{old} \neq n'$
7. $sub(p_{old}, p_{new}, \emptyset) = \emptyset$,

where p_{old}*,* p_{new} *and* p *are places,* t *is a transition,* n *and* n' *are nodes of a railroad net,* e *is an expression,* S *is a set of pairs of places,* P *is a set of places,* T *is a set of transitions,* A *is a set of arcs.*

The pair replacement function (sub^P) takes a set of pairs of places, and a colored Petri net as arguments. For each pair of places it will generate the union place of the two places, and replace the two original places with this new union place.

Definition 36 *The pair replacement function* (sub^P) *is defined by the equations:*

1. $sub^P(\{\langle p, p'\rangle\} \cup S, \langle P, T, A\rangle) =$
   $sub^P(S \backslash \{\langle p, p'\rangle\}, \langle sub(p', p \textcircled{p} p', sub(p, p \textcircled{p} p', \langle P, T, A\rangle))\rangle)$

2. $sub^P(\emptyset, \langle P, T, A\rangle) = \langle P, T, A\rangle$,

where p and p' are places, S is a set of pairs of places, P is a set of places, T is a set of transitions, A is a set of arcs.

Definition 37 *The interface union ($\uplus$) of two interfaces i_1 and i_2 are defined by the following equations:*

1. $i_1 \uplus i_2 = \{i_1\} \cup \{i_2\}$ *if* $i_1 \neq I_\emptyset$ *and* $i_2 \neq I_\emptyset$
2. $i_1 \uplus i_2 = I_\emptyset$ *if* $i_1 = I_\emptyset$ *or* $i_2 = I_\emptyset$

Definition 38 *Let $N_1 = \langle \mathrm{CPN}_1, \mathrm{I}_1\rangle$ and $N_2 = \langle \mathrm{CPN}_2, \mathrm{I}_2\rangle$ be two joinable railroad nets over the binding $b = [i_1, i_2]$. Then the composition of N_1 and N_2 are given by:*

$$N_1 \bowtie_b N_2 \quad = \quad \langle sub^P(i_1 \boxed{\mathbb{p}}\, i_2, \mathrm{CPN}_1 \sqcup \mathrm{CPN}_2), (\mathrm{I}_1 \cup \mathrm{I}_2) \backslash (i_1 \uplus i_2)\rangle$$

Note that the two joined interfaces are removed from the set of interfaces. Consequently an interface can only be joined with one other interface.

Lemma 3 *The composition of two joinable railroad nets is a partial abelian monoid.*

Proof: In order to prove that the composition of two joinable railroad nets is a partial abelian monoid, we must prove that it satisfies the laws of closure, identity, associativity and commutativity.

Closure: If N_1 and N_2 are two joinable railroad nets over the binding b then $N_1 \bowtie_b N_2$ is a railroad net. This follows from the restriction joinable.

Identity: We must prove that there exists an identity element $N_\emptyset$ such that $N_1 \bowtie_b N_\emptyset = N_1 = N_\emptyset \bowtie_b N_1$, where $b = [i_1, I_\emptyset]$, and i_1 is an interface in N_1

$$\begin{aligned} N_1 \bowtie_b N_\emptyset &= \langle sub^P(i_1 \boxed{\mathbb{p}}\, I_\emptyset, \mathrm{CPN}_1 \sqcup \mathrm{CPN}_\emptyset), \mathrm{I}_1 \cup I_\emptyset \backslash (i_1 \uplus I_\emptyset)\rangle \\ &= \langle sub^P(\{\langle p_1, P_\emptyset\rangle, \ldots, \langle p_n, P_\emptyset\rangle\}, \mathrm{CPN}_1), \mathrm{I}_1 \backslash I_\emptyset\rangle \\ &\overset{1}{=} \langle \mathrm{CPN}_1, \mathrm{I}_1\rangle \\ &= N_1 \end{aligned}$$

Equation 1 holds because the pair replacement $sub^P(\{\langle p_1, P_\emptyset\rangle, \ldots, \langle p_n, P_\emptyset\rangle\}, \mathrm{CPN}_1)$ is equal to the substitution $sub(P_\emptyset, p_i \textcircled{p} P_\emptyset, sub(p_i, p_i \textcircled{p} P_\emptyset, \mathrm{CPN}_1))$ recursively for all p_i in i_1. The expression $p_i \textcircled{p} P_\emptyset$ is always equal to p_i, and then the substitution will be $sub(P_\emptyset, p_i, sub(p_i, p_i, \mathrm{CPN}_1))$, which is equal to CPN_1.

$$\begin{aligned}
N_1 \bowtie_b N_\emptyset &= \langle sub^P(i_1 \boxed{\mathfrak{p}}\, I_\emptyset, \mathrm{CPN}_1 \sqcup \mathrm{CPN}_\emptyset), \mathrm{I}_1 \cup I_\emptyset \backslash (i_1 \uplus I_\emptyset)\rangle \\
&\overset{1}{=} \langle sub^P(I_\emptyset \boxed{\mathfrak{p}}\, i_1, \mathrm{CPN}_\emptyset \sqcup \mathrm{CPN}_1), I_\emptyset \cup \mathrm{I}_1 \backslash (I_\emptyset \uplus i_1)\rangle \\
&= N_\emptyset \bowtie_\epsilon N_1
\end{aligned}$$

Equation 1 holds because:

- $i_1 \boxed{\mathfrak{p}}\, I_\emptyset = I_\emptyset \boxed{\mathfrak{p}}\, i_1$ by equation number 5 of Definition 34.
- $\mathrm{CPN}_1 \sqcup \mathrm{CPN}_\emptyset = \mathrm{CPN}_\emptyset \sqcup \mathrm{CPN}_1$ because the flat union is commutative.
- $\mathrm{I}_1 \cup I_\emptyset = I_\emptyset \cup \mathrm{I}_1$ because the union operator is commutative.
- $i_1 \uplus I_\emptyset = I_\emptyset \uplus i_1$ because $\uplus$ is commutative.

Associativity: We must show that: $(N_1 \bowtie_{b_1} N_2) \bowtie_{b_2} N_3 = N_1 \bowtie_{b_1} (N_2 \bowtie_{b_2} N_3)$.

$$\begin{aligned}
& (N_1 \bowtie_{b_1} N_2) \bowtie_{b_2} N_3 \\
= \; & (\langle sub^P(i_1 \boxed{\mathfrak{p}}\, i_2, \mathrm{CPN}_1 \sqcup \mathrm{CPN}_2), (\mathrm{I}_1 \cup \mathrm{I}_2) \backslash (i_1 \uplus i_2)\rangle) \bowtie_{b_2} N_3 \\
= \; & \langle sub^P(i_{2'} \boxed{\mathfrak{p}}\, i_3, \mathrm{CPN}_3 \sqcup (sub^P(i_1 \boxed{\mathfrak{p}}\, i_2, \mathrm{CPN}_1 \sqcup \mathrm{CPN}_2))), \\
& (((\mathrm{I}_1 \cup \mathrm{I}_2) \backslash (i_1 \uplus i_2)) \cup \mathrm{I}_3) \backslash (i_{2'} \uplus i_3)\rangle \\
\overset{1}{=} \; & \langle sub^P(i_{2'} \boxed{\mathfrak{p}}\, i_3, sub^P(i_1 \boxed{\mathfrak{p}}\, i_2, \mathrm{CPN}_3 \sqcup (\mathrm{CPN}_1 \sqcup \mathrm{CPN}_2))), \\
& (((\mathrm{I}_1 \cup \mathrm{I}_2) \cup \mathrm{I}_3) \backslash (i_1 \uplus i_2)) \backslash (i_{2'} \uplus i_3)\rangle \\
\overset{2}{=} \; & \langle sub^P(i_1 \boxed{\mathfrak{p}}\, i_2, sub^P(i_{2'} \boxed{\mathfrak{p}}\, i_3, \mathrm{CPN}_3 \sqcup (\mathrm{CPN}_1 \sqcup \mathrm{CPN}_2))), \\
& (((\mathrm{I}_1 \cup \mathrm{I}_2) \cup \mathrm{I}_3) \backslash (i_{2'} \uplus i_3)) \backslash (i_1 \uplus i_2)\rangle \\
\overset{3}{=} \; & \langle sub^P(i_1 \boxed{\mathfrak{p}}\, i_2, sub^P(i_{2'} \boxed{\mathfrak{p}}\, i_3, \mathrm{CPN}_1 \sqcup (\mathrm{CPN}_2 \sqcup \mathrm{CPN}_3))), \\
& (((\mathrm{I}_2 \cup \mathrm{I}_3) \cup \mathrm{I}_1) \backslash (i_{2'} \uplus i_3)) \backslash (i_1 \uplus i_2)\rangle \\
\overset{4}{=} \; & \langle sub^P(i_1 \boxed{\mathfrak{p}}\, i_2, \mathrm{CPN}_1 \sqcup (sub^P(i_{2'} \boxed{\mathfrak{p}}\, i_3, \mathrm{CPN}_2 \sqcup \mathrm{CPN}_3))), \\
& (((\mathrm{I}_2 \cup \mathrm{I}_3) \backslash (i_{2'} \uplus i_3)) \cup \mathrm{I}_1) \backslash (i_1 \uplus i_2)\rangle \\
= \; & N_1 \bowtie_{b_1} \langle sub^P(i_{2'} \boxed{\mathfrak{p}}\, i_3, \mathrm{CPN}_2 \sqcup \mathrm{CPN}_3), (\mathrm{I}_2 \cup \mathrm{I}_3) \backslash (i_{2'} \uplus i_3)\rangle \\
= \; & N_1 \bowtie_{b_1} (N_2 \bowtie_{b_2} N_3)
\end{aligned}$$

Equation 1 is valid because none of the places in i_1 or i_2 are present in CPN_3 and because neither i_1 or i_2 are elements in I_3. Equation 2 holds because none of the places in i_1 or i_2 are present in $i_{2'}$ or i_3. Therefore it does not matter what order they are substituted in. The same argument holds for the subtractions of interfaces. In equation 3, the associativity of the union operator, ($\cup$), and the flat union operator ($\sqcup$) is used. Because none of the places in the interfaces $i_{2'}$ or i_3 are present in CPN_1 or in I_1 then equation 4 is valid.

Commutativity: We must show that: $N_1 \bowtie_b N_2 = N_2 \bowtie_b N_1$

$$\begin{aligned}
N_1 \bowtie_b N_2 &= \langle sub^P(i_1 \boxed{\mathfrak{p}}\, i_2, \mathrm{CPN}_1 \sqcup \mathrm{CPN}_2), (\mathrm{I}_1 \cup \mathrm{I}_2)\backslash(i_1 \uplus i_2)\rangle \\
&\overset{1}{=} \langle sub^P(i_1 \boxed{\mathfrak{p}}\, i_2, \mathrm{CPN}_2 \sqcup \mathrm{CPN}_1), (\mathrm{I}_2 \cup \mathrm{I}_1)\backslash(i_1 \uplus i_2)\rangle \\
&\overset{2}{=} \langle sub^P(i_1 \boxed{\mathfrak{p}}\, i_2, \mathrm{CPN}_2 \sqcup \mathrm{CPN}_1), (\mathrm{I}_2 \cup \mathrm{I}_1)\backslash(i_2 \uplus i_1)\rangle \\
&\overset{3}{=} \langle sub^P(i_2 \boxed{\mathfrak{p}}\, i_1, \mathrm{CPN}_2 \sqcup \mathrm{CPN}_1), (\mathrm{I}_2 \cup \mathrm{I}_1)\backslash(i_2 \uplus i_1)\rangle \\
&= N_2 \bowtie_b N_1
\end{aligned}$$

Equation 1 is valid because of the commutativity of the union operator ($\cup$) and the flat union operator ($\sqcup$). Equation 2 is valid because $\uplus$ is commutative. Equation 3 holds because:

$$\begin{aligned}
&sub^P(i_1 \boxed{\mathfrak{p}}\, i_2, \mathrm{CPN}_2 \sqcup \mathrm{CPN}_1) = \\
&sub^P(\{\langle p_{1_1}, p_{2_1}\rangle, \ldots, \langle p_{1_n}, p_{2_n}\rangle\}, \mathrm{CPN}_2 \sqcup \mathrm{CPN}_1) = \\
&sub(p_{2_n}, p_{1_n} \textcircled{\mathfrak{p}} p_{2_n}, sub(p_{1_n}, p_{1_n} \textcircled{\mathfrak{p}} p_{2_n}, \ldots, \\
&sub(p_{2_1}, p_{1_1} \textcircled{\mathfrak{p}} p_{2_1}, sub(p_{1_1}, p_{1_1} \textcircled{\mathfrak{p}} p_{2_1}, \mathrm{CPN}_2 \sqcup \mathrm{CPN}_1)))) \overset{1}{=} \\
&sub(p_{2_n}, p_{2_n} \textcircled{\mathfrak{p}} p_{1_n}, sub(p_{1_n}, p_{2_n} \textcircled{\mathfrak{p}} p_{1_n}, \ldots, \\
&sub(p_{2_1}, p_{2_1} \textcircled{\mathfrak{p}} p_{1_1}, sub(p_{1_1}, p_{2_1} \textcircled{\mathfrak{p}} p_{1_1}, \mathrm{CPN}_2 \sqcup \mathrm{CPN}_1)))) \overset{2}{=} \\
&sub(p_{1_n}, p_{2_n} \textcircled{\mathfrak{p}} p_{1_n}, sub(p_{2_n}, p_{2_n} \textcircled{\mathfrak{p}} p_{1_n}, \ldots, \\
&sub(p_{1_1}, p_{2_1} \textcircled{\mathfrak{p}} p_{1_1}, sub(p_{2_1}, p_{2_1} \textcircled{\mathfrak{p}} p_{1_1}, \mathrm{CPN}_2 \sqcup \mathrm{CPN}_1)))) = \\
&sub^P(i_2 \boxed{\mathfrak{p}}\, i_1, \mathrm{CPN}_2 \sqcup \mathrm{CPN}_1)
\end{aligned}$$

Equation 1 holds because $\textcircled{\mathfrak{p}}$ is commutative. Equation 2 is valid because the substitution of nodes are commutative. ♣

As for specification graphs we need a copy function ($\mathscr{C}$), which makes a copy of a railroad net, and replaces all ids with fresh ones.

Chapter 5

Refinement of railroad nets

The atomic railroad nets shown in Section 4.2 have a very simple behavior. In order to get the behavior of the models more complex, we introduce several *refinement functions*. A refinement function is defined from a library of railroad nets to another. At least one railroad net of a refined library is larger than its ancestor. By larger we mean that it has more places, transition, arcs or tokens, or that some expressions are expanded. For a more definition of refinement see [6].

5.1 Safety

One problem with the basic models is that trains may pass each other on a single track. This is not a life like behavior as an attempt to do so would result in a crash. To avoid this the *noTrain token* is introduced. If a place has a noTrain token this means that the place is unoccupied. In the initial marking, all move places should either have a train token or a noTrain token. Figure 5.1 shows the *safe track segment*. In order for the train in the `Left` place to move to the `Right` place, by the firing of the `move+` transition, there has to be a noTrain token in the `Right` place. Note that after the train has moved to the `Right` place, the `Left` place will contain a noTrain token, and is thereby unoccupied. This situation is displayed in Figure 5.2. The result is that trains can not pass each other on a single track. Another result is that two trains can not occupy the same place. All the atomic components are refined in the same way as the track component. For figures of all safe

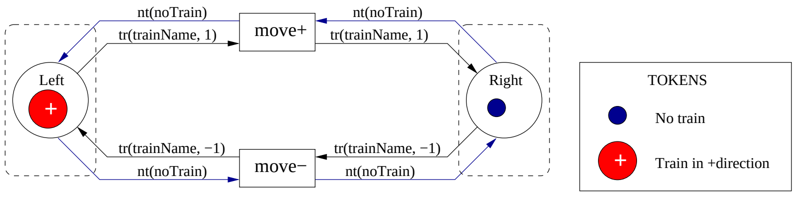


Figure 5.1: The safe track segment

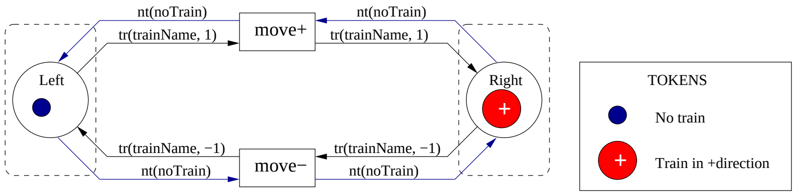


Figure 5.2: The safe track after firing

atomic components, see [25].

5.2 Train routes and train controlled turnouts

When a train has reached a turnout component, or a component that contains a turnout, the route it has been taking is dependent on the current state of the turnout. Fortunately this is not the case in real railway systems. The trains should have a destination. To do this all turnouts must be identifiable, so the train know where it is. This is solved by adding an *id place* to all turnouts. These id places has a token that carries the unique id of the turnout. Figure 5.3 shows such a turnout component, and the id place is the purple colored place at the bottom. To keep track of which way a train should go in a turnout, it is equipped with a *train route* or *travel plan*. A travel plan is a set of pairs of switch ids and branches, left or right. Note that the two transitions that moves a train from the `Left` place to the `UpperRight` and `LowerRight` places are equipped with guards. The guard $[(id, left)]$ evaluates to true if the pair $(id, left)$ is in the train route of the train. The term *id* is the id of this turnout, and is given by the token in the `SwitchId` place. When a train moves from one of the right hand places to the left hand place it does not have to check its train route as it is only one

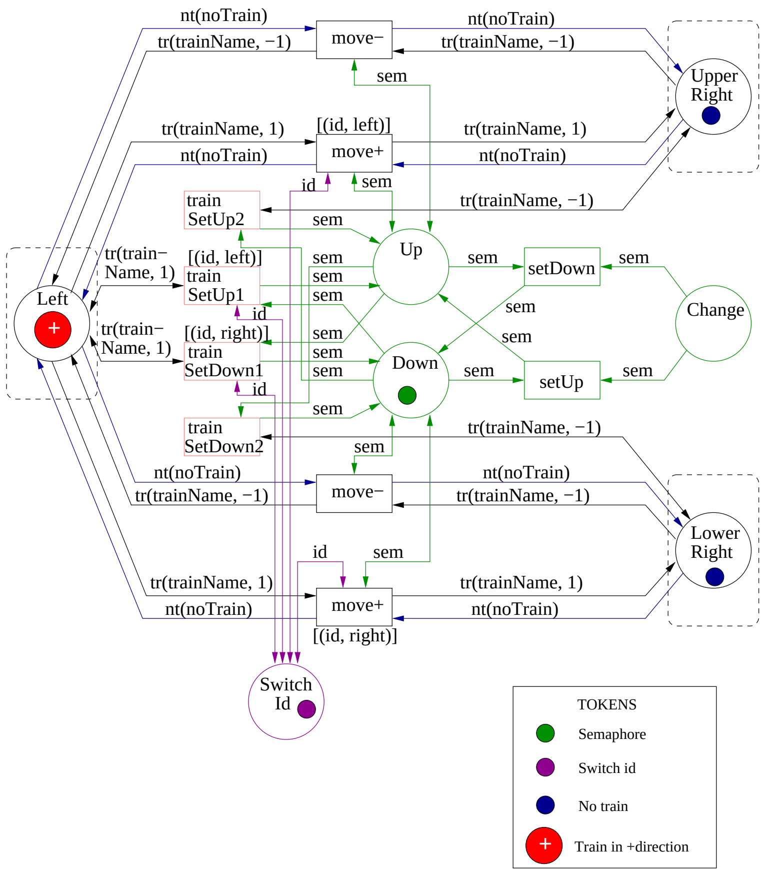


Figure 5.3: The turnout with train routes

possible path.

As described in Section 4.2.4 the component might have dead-lock situation if train that travels in the positive direction is in the `UpperRight1` place, and the semaphore token of the point machine is in the `Down` place. Introducing the guards results in a new dead-lock possibility to the turnout component. The new dead-lock situation arises when it is a train in the `Left` place that travels in the positive direction and is supposed to turn left in the turnout, and the semaphore token of the point machine is in the `Down` place. This is the marking shown in Figure 5.3. As is the case for the dead-lock described in Section 4.2.4, the mirror image of the new dead-lock is also a dead-lock. The four newly introduced pink transitions in Figure 5.3 solves this problem. When a train enters a turnout in the wrong state, it will change the state of the turnout. If the train in Figure 5.3 is supposed to turn left in this turnout, the `trainSetUp1` transition is enabled. A firing of this transition results in that the semaphore token is moved from the `down` place to the `up` place. Then the train can move to the `UpperRight` place. The two transitions `trainSetDown2` and `trainSetUp2` solves the dead-lock problem described in Section 4.2.4 in a similar way.

A problem with this approach is that live-locks may occur. Figure 5.4 shows the marking in a turnout causing a possible live-lock. In this marking the upper `move-` transition, and the `trainSetDown2` transition are enabled. If the *move* transition fires, the upper train moves to the `Left` place, and the live-lock is resolved. If the `trainSetDown2` transition fires, the state of the turnout is changed. Now the `trainSetUp2` and the lower `move-` are enabled. Again the firing of the move transition solves the problem, but the firing of the lower `trainSetUp2` results in the marking in Figure 5.4 again.

The live-lock is resolved assuming *strong fairness*: if a transition t is enabled infinitely many times, t will fire infinitely many times. This ensures that one of the move transitions eventually fires and resolves the live-lock. In Maude the fairness is obtained by using the `frewrite` command. The live-lock could be eliminated by letting the train change the state of the turnout and move in the same operation. However, we keep the two operations separate in the model, since this is closer to how real trains would operate.

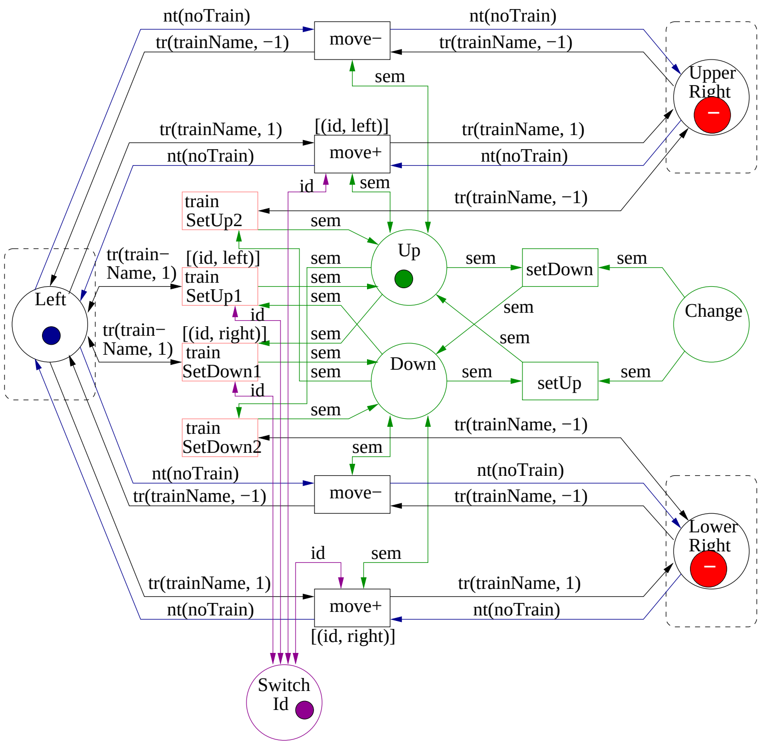


Figure 5.4: The live-lock situation

5.3 How to implement time in a railway system

The easiest way of introducing time to the models is to give the transitions fixed delays. This means that it always takes the same amount of time for a train to move from one point to another. This delay can then either be

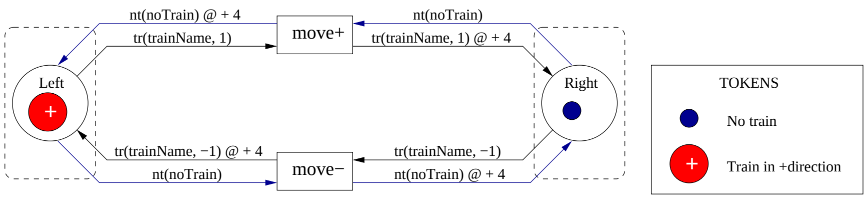


Figure 5.5: The track segment with constant time

fixed for a specific kind of component or be an average time found by empiric tests. The latter will be closer to reality, but also make the generation of the nets more time consuming as it no longer can be done automatically. If such data does not exists it is an extensive task getting them. Figure 5.5 shows the track component with a constant delay of four time units. Note that the noTrain tokens are delayed as well. This means that both places are occupied while the train is between them.

To make it more realistic we can add or subtract a factor to the constant. This factor can either be random or be dependent of physical factors or even a combination. Such physical factors may be the engine driver's profile, the trains profile, the passenger load, the weather conditions etc. The driver profile can contain information on different characteristics of the driver. Some examples could be aggressiveness, will to exceed the speed limit, and will to wait for passengers. The train profile may contain specifications of the trains, such as breaking effect, acceleration, top speed, effect of passenger load etc.

5.4 Sensitive tracks with separate signals

A different approach to introduce time to the system, is to calculate how long time it takes to drive a stretch. The figures 5.6 and 5.7 shows the track component and the turnout with such calculations. To be able to do these calculations, some information is needed. Each stretch will be equipped with a *info place*. This place contains a *info token*. The info token has information on the length of the stretch. The info places are the turquoise colored places of figures 5.6 and 5.7.

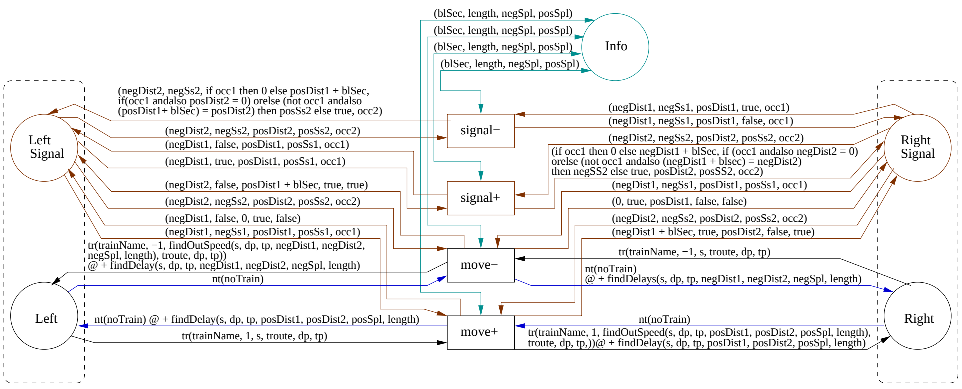

Info
(blSec, length, negSpl, posSpl)
(blSec, length, negSpl, posSpl)
(blSec, length, negSpl, posSpl)
(blSec, length, negSpl, posSpl)
Left Signal
Right Signal
Left
Right
signal−
signal+
move−
move+
(negDist2, negSs2, if occ1 then 0 else posDist1 + blSec, if(occ1 andalso posDist2 = 0) orelse (not occ1 andalso (posDist1+ blSec) = posDist2) then posSs2 else true, occ2)
(negDist2, negSs2, posDist2, posSs2, occ2)
(negDist1, false, posDist1, posSs1, occ1)
(negDist1, true, posDist1, posSs1, occ1)
(negDist2, false, posDist1 + blSec, true, true)
(negDist2, negSs2, posDist2, posSs2, occ2)
(negDist1, false, 0, true, false)
(negDist1, negSs1, posDist1, posSs1, occ1)
tr(trainName, −1, findOutSpeed(s, dp, tp, negDist1, negDist2, negSpl, length), troute, dp, tp)) @ + findDelay(s, dp, tp, negDist1, negDist2, negSpl, length)
nt(noTrain)
nt(noTrain) @ + findDelay(s, dp, tp, posDist1, posDist2, posSpl, length)
tr(trainName, 1, s, troute, dp, tp)
(negDist1, negSs1, posDist1, true, occ1)
(negDist1, negSs1, posDist1, false, occ1)
(negDist2, negSs2, posDist2, posSs2, occ2)
(if occ1 then 0 else negDist1 + blSec, if (occ1 andalso negDist2 = 0) orelse (not occ1 andalso (negDist1 + blsec) = negDist2) then negSS2 else true, posDist2, posSS2, occ2)
(negDist1, negSs1, posDist1, posSs1, occ1)
(0, true, posDist1, false, false)
(negDist2, negSs2, posDist2, posSs2, occ2)
(negDist1 + blSec, true, posDist2, false, true)
tr(trainName, −1, s, troute, dp, tp)
nt(noTrain) @ + findDelays(s, dp, tp, negDist1, negDist2, negSpl, length)
nt(noTrain)
tr(trainName, 1, findOutSpeed(s, dp, tp, posDist1, posDist2, posSpl, length), troute, dp, tp,))@ + findDelay(s, dp, tp, posDist1, posDist2, posSpl, length)

Figure 5.6: The track segment with separate signals

In many railroad systems the speed limits depend upon how many block sections there are to the next obstacle. So is the case for the subway in Oslo. An obstacle could be another train, a turnout in the wrong state etc. The info token therefore contains two lists of speed limits, one for each direction. The position in the list represents how many free block sections needed for this to be the actual speed limit. an example of one such speed limit list could be:

speed limit list	[	15	,	15	,	30	,	50	,	80	]
numbers of free block sections		0		1		2		3		≥ 4	

From the example above we can read that if there are two free block sections, the speed limit is 30 kilometer per hour. The info token also has a slot for information on whether the two ends of the stretch is in the same block section. This slot carries the value one if the ends are i different block sections and zero if they are in the same

The real difficulty is on how to keep track of how many free block sections there are in each direction. We still want to obtain modularity so a problem arise. We need to access information on the situation in other parts of the net. A solution to the problem is to introduce a stream of signals with information on how far it is to the next obstacle in each direction. This stream runs

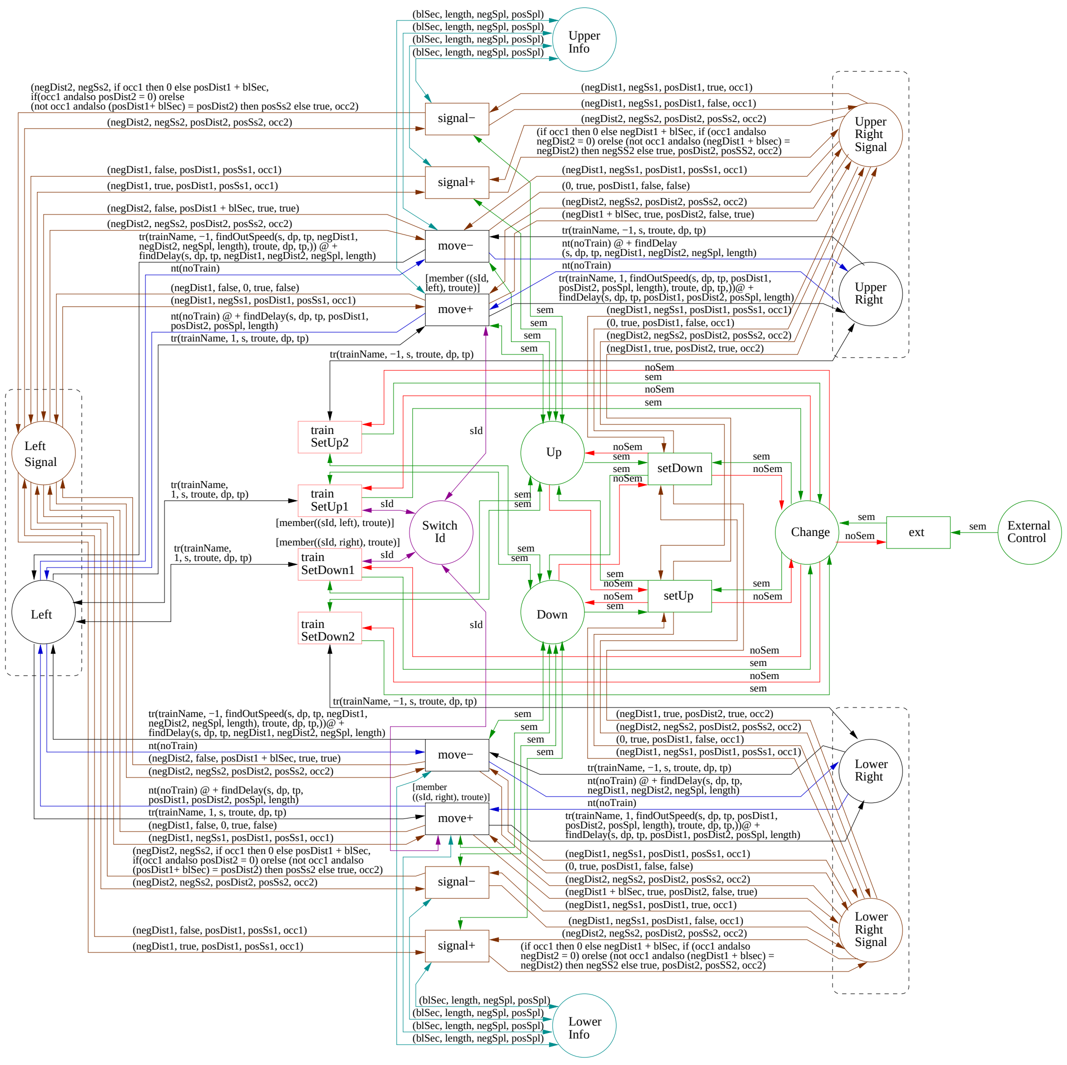

Upper Info
Upper Right Signal
Upper Right
signal−
signal+
move−
move+
Left Signal
Left
train SetUp2
train SetUp1
train SetDown1
train SetDown2
Switch Id
Up
Down
setDown
setUp
Change
ext
External Control
move−
move+
signal−
signal+
Lower Right
Lower Right Signal
Lower Info
(blSec, length, negSpl, posSpl)
(negDist2, negSs2, if occ1 then 0 else posDist1 + blSec, if(occ1 andalso posDist2 = 0) orelse (not occ1 andalso (posDist1+ blSec) = posDist2) then posSs2 else true, occ2)
(negDist2, negSs2, posDist2, posSs2, occ2)
(negDist1, false, posDist1, posSs1, occ1)
(negDist1, true, posDist1, posSs1, occ1)
(negDist2, false, posDist1 + blSec, true, true)
tr(trainName, −1, findOutSpeed(s, dp, tp, negDist1, negDist2, negSpl, length), troute, dp, tp,)) @ + findDelay(s, dp, tp, negDist1, negDist2, negSpl, length)
nt(noTrain)
(negDist1, false, 0, true, false)
(negDist1, negSs1, posDist1, posSs1, occ1)
nt(noTrain) @ + findDelay(s, dp, tp, posDist1, posDist2, posSpl, length)
tr(trainName, 1, s, troute, dp, tp)
(negDist1, negSs1, posDist1, true, occ1)
(negDist1, negSs1, posDist1, false, occ1)
(if occ1 then 0 else negDist1 + blSec, if (occ1 andalso negDist2 = 0) orelse (not occ1 andalso (negDist1 + blsec) = negDist2) then negSS2 else true, posDist2, posSS2, occ2)
(0, true, posDist1, false, false)
(negDist1 + blSec, true, posDist2, false, true)
tr(trainName, −1, s, troute, dp, tp)
nt(noTrain) @ + findDelay (s, dp, tp, negDist1, negDist2, negSpl, length)
tr(trainName, 1, findOutSpeed(s, dp, tp, posDist1, posDist2, posSpl, length), troute, dp, tp,))@ + findDelay(s, dp, tp, posDist1, posDist2, posSpl, length)
(0, true, posDist1, false, occ1)
(negDist1, true, posDist2, true, occ2)
[member ((sId, left), troute)]
[member((sId, left), troute)]
[member((sId, right), troute)]
[member ((sId, right), troute)]
sId
sem
noSem
tr(trainName, 1, s, troute, dp, tp)
(negDist1, true, posDist2, true, occ2)
(if occ1 then 0 else negDist1 + blSec, if (occ1 andalso negDist2 = 0) orelse (not occ1 andalso (negDist1 + blsec) = negDist2) then negSS2 else true, posDist2, posSS2, occ2)

Figure 5.7: The turnout with separate signals

parallel to the trains. For each track place there is one corresponding *signal place*. Note that these to corresponding places are members of the same interface. When something in the system changes, whether it is a train moving or a point machine changing the state of a turnout, the signals must be updated. The signals are modeled by *signal tokens* which exchange values with its neighbors. Each signal token contains five values. `negDistance` and `posDistance` is the number of block sections to the nearest obstacle in each direction. `negSendSignal` and `posSendSignal` is an instruction on whether to send the values to its neighbor or not. The value `occupied` holds information on whether there is a train in the corresponding track place. If `negSendSignal` is true this means that the signal token must send its `negDistance` to its neighbor in the positive direction. Trains must inform the signal tokens of their movement.

5.4.1 The signals

Train movement

When a train moves it sends a signal in both directions. For instance, a train moves from the place `Left` to the `Right` place in the track component of Figure 5.6. Then the signal tokens from the two signal places are consumed, and two new ones are added. The token added to the place `LeftSignal` have the same `negDistance` value as the one that was here before. There is no use sending this value in the positive direction since the token generated there already has seen this value. Hence the `negSendSignal` is set to false. The `posDistance` is set to zero as we know that there is a train in the `Right` place. This value is new to us and must be sent in the negative direction. Therefore the `posSendSignal` is set to true. The place `Left` gets a noTrain token and is no longer occupied so the `occupied` is set to false. The token added to the place `RightSignal` gets the following values. The `negDistance` is set to the value `negDistance` of the token in the place `LeftSignal` if `Left` and `Right` are in the same block section, and one more than that if they are not. This value is new so it must be passed on, hence the `negSendSignal` is set to true. The `posDistance` has not changed and is set to the same as it was. It is no use sending this value in the negative direction so the `posSendSignal` is set to false. The `occupied` is obviously set to true.

The signal stream

When a signal token in the place `LeftSignal`, of the track segment in Figure 5.6, has true as its `negSendSignal` value this means that it is supposed to send it's `negDistance` value in the positive direction. This is handled by the transition `signal+`. This transition consumes the signal tokens from the two signal places. To the place `LeftSignal` it adds a token equal to the one it consumed but with the `negSendSignal` set to false, because it is no point in sending the same signal again. A token is added to the place `RightSignal` as well. This token has the same `posDistance`, `posSendSignal` and `occupied` values as the one that where consumed from here. The value of `negDistance` is updated if necessary. If the `occupied` of the token in `LeftSignal` is true (this means that there is a train in the place `Left`), then `negDistance` is set to zero. If `occupied` is false then `negDistance` is set to the `negDistance` of the token from `LeftSignal` if `Left` and `Right` are in the same block section, if not it it set to one more than the `negDistance` of the token from `LeftSignal`. If the new value of `negDistance` is the same as it was before, then the value of `negSendSignal` is not changed. This is to prevent signaling the same value over and over again. But if the value of `negDistance` has changed then the value of `negSendSignal` is set to true, we have to send this value further in the positive direction.

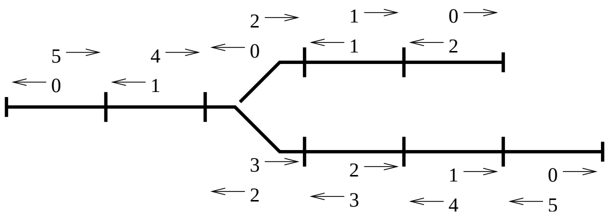
Note: The signal tokens are not timed. This means that signals travel infinitely faster than trains. This is of course a simplification of reality, but we find it a reasonable one.

Changing the state of a turnout

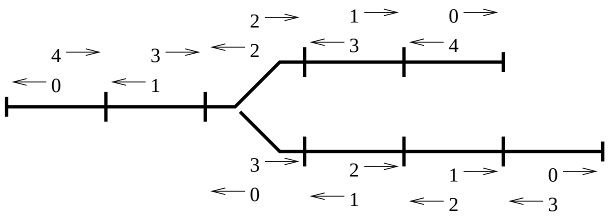
Figure 5.8 shows two states of a tiny railway system. The tracks are divided into block sections. The numbers shows the number of free block sections in each direction.

In the system of Figure 5.8(a), the position of the turnout is directed downwards, hence the broken line on the upper branch. Note that in the upper branch the turnout is regarded as an obstacle. If the state of the turnout is changed, all signals must be updated. This will result in the situation of Figure 5.8(b).

In the turnout of Figure 5.7 the updating of the signals is performed, when



(a) before state change



(b) after state change

Figure 5.8: Free block sections

the transitions `setUp` and `setDown` fires. If for instance `setUp` fires, the turnout is set upwards. The signal tokens of `LeftSignal` and `RightSignal` are consumed, and new ones are added. The signal token created in the `LeftSignal` place gets the `negDistance` value of the token from the `RightSignal` place. In Figure 5.8 the value is the number 2. This is a new value so it must be passed on. Hence `negSendSignal` is set to true. The distance to the next obstacle in the positive direction is the same as before, but the value must be sent in the negative direction so `posSendSignal` is set to be true. In the lower branch the distance to the next obstacle in the negative direction is now zero. therefore the `negDistance` value of the signal token created in the `RightSignal` place is zero. This value must be passed on so `negSendSignal` is set to true.

5.4.2 The updated point machine

As described in Section 5.4.1 the signals must be updated when the state of a turnout changes. The update happens when the transition `setUp` or `setDown` fires. It is therefore important that the state of the turnout never changes unless one of these transitions fire. This is not the case of the turnout in Figure 5.3. When one of the pink transition fires, the state of the turnout is changed without the firing of `setUp` or `setDown`. The point machine of the turnout in Figure 5.7 is updated to handle this problem. Instead of consuming a semaphore token from one of the places `Up` or `Down` and adding it to the other one, the pink transitions now changes the state of the turnout by adding a token to the `Change` place. The state of the turnout will then be changed by the firing of `setUp` or `setDown`. To avoid the possibility of an accumulation of semaphore tokens in the `Change` place, a *non-semaphore token* is introduced. The presence of a non-semaphore token in a place, mens that there is no semaphore token in it. All transitions that generates a semaphore token to the `Change` place, removes a non-semaphore token at the same time. The same goes for the places `Up` and `Down`. We still want to be able to operate the turnout manually by adding a semaphore token, without having to remove a non-semaphore token. This possibility is achieved by adding the place `ExternalControl` and the transition `ext`.

5.4.3 Calculation of speed and delays

To be able to predict the time it take to get from a place A to a place B by a transition T, we need to establish a driving pattern. All move transitions must have information on the speed limits in both directions. When we later on refer to the speed limit of A, we mean the speed limit in point A in the direction of point B. Further when we say the speed limit of B, This means the speed limit of point B in the same direction. For simplicity reasons we will only consider linear acceleration and retardation.

Figure 5.9 shows a flow chart of the algorithm we use to calculate how long time it takes to move through a component. The flow chart consists of squares which represent activities or tasks, diamonds which represent decision points, and arrows which represent flow of control. The ovals are the start and end points, and delays are represented by four sided shapes with a curved right edge.

If the train has been speeding and not yet performed the safety stop, or if the train is speeding then the train must stop as soon as possible. It must also stand still for a period of time, before it can accelerate toward the speed limit again. Otherwise if the speed limit of A is smaller than or equal to the speed limit of B then the train will try to accelerate to the speed limit of A, if it does not already travel at that speed, and then keep this speed all the way to point B. Furthermore if the speed of the train is greater than the speed limit of B then if the distance between A and B is to short for the train to break down to the speed limit of B by the time it gets to B then the train will break at max break power for the entire distance. Otherwise if the distance is to small for the train to reach the speed limit of B by normal retardation. Then the train brakes just as hard as it has to to reach the speed limit of B. Otherwise the train accelerates until it reaches the speed limit of A or for as long as it can before it performs a normal retardation and reaches the speed limit of B just as it gets to point B.

Hence time a train uses to get from A to B is dependent on the following factors:

- Speed at point A
- Speed limit at point A
- Speed limit at point B

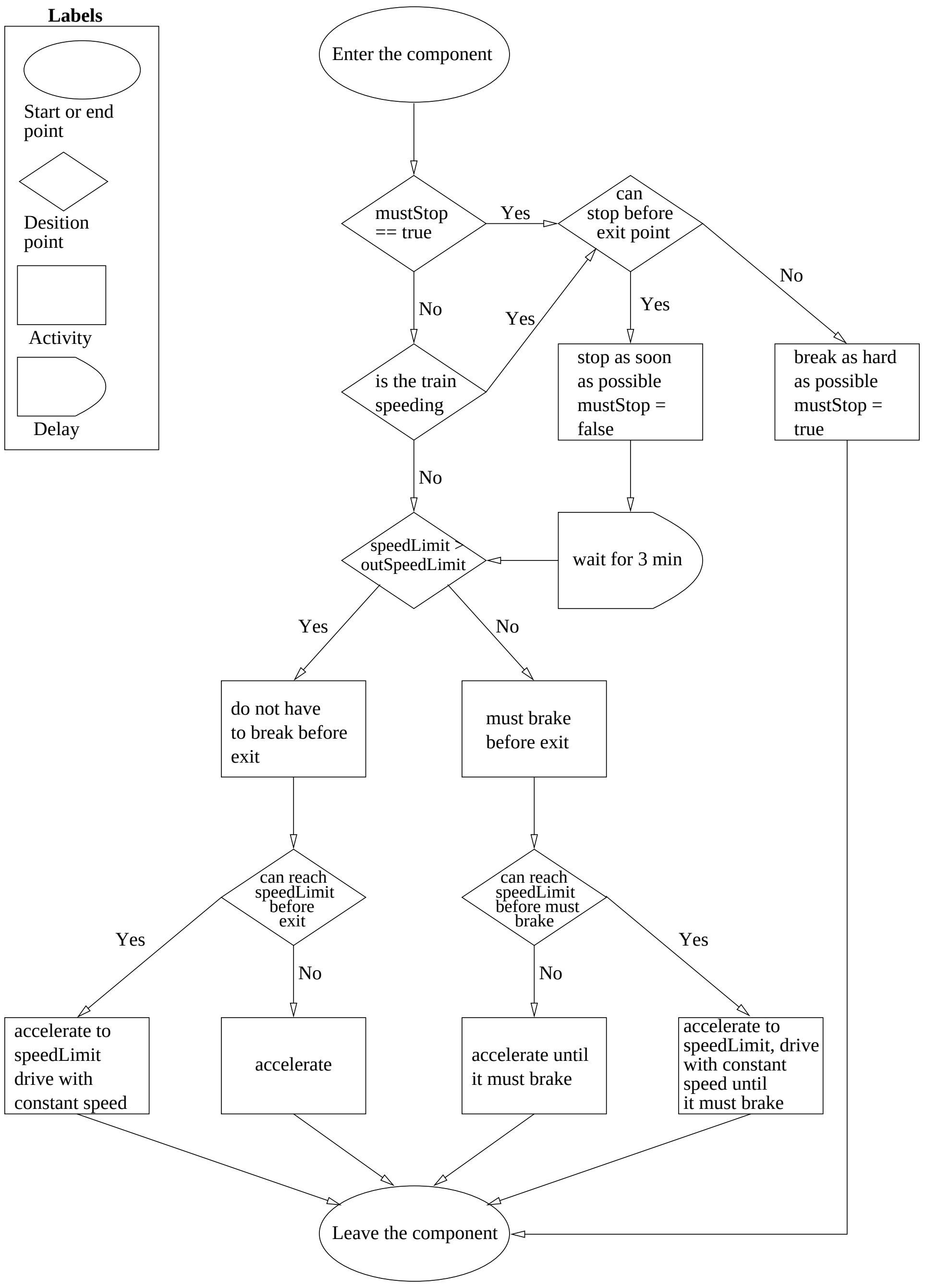

Labels
Start or end point
Desition point
Activity
Delay
Enter the component
mustStop == true
Yes
can stop before exit point
No
Yes
No
Yes
is the train speeding
stop as soon as possible mustStop = false
break as hard as possible mustStop = true
No
speedLimit > outSpeedLimit
wait for 3 min
Yes
No
do not have to break before exit
must brake before exit
can reach speedLimit before exit
can reach speedLimit before must brake
Yes
No
No
Yes
accelerate to speedLimit drive with constant speed
accelerate
accelerate until it must brake
accelerate to speedLimit, drive with constant speed until it must brake
Leave the component

Figure 5.9: Driving pattern of the trains

- The acceleration of the train
- The normal retardation of the train
- The maximum retardation of the train

The ML-functions used to calculate the delays may be found on the attached CD-ROM. Note that there is a difference between normal retardation and maximum retardation. We have decided that the normal retardation is dependent on how aggressive the driver is, and that it is somewhere between 0.5 and 1 times the maximum retardation. In the driver profiles the aggressiveness is represented by a number between zero and one hundred. The normal retardation must then be:

$$\text{normal retardation} = \left(\frac{aggressiveness}{200} + 0.5\right) * \text{max retardation}$$

The acceleration is dependent on the drivers aggressiveness in the same way:

$$\text{acceleration} = \left(\frac{aggressiveness}{200} + 0.5\right) * \text{max acceleration}$$

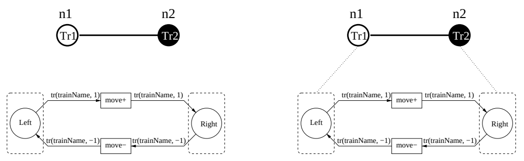
These expressions can of course be altered if empiric tests shows that the retardation or acceleration varies more, or less, than we have presumed.

Chapter 6

Saturation and translation

6.1 Saturation

The process of connecting the specifications to the railroad nets are called saturation. When saturating a specification, all template graphs are connected to a railroad component. connecting a specification graph to a railroad net means associating all interface nodes of the specification graph with different interfaces of the railroad net.



(a) Before saturation

(b) After saturation

Figure 6.1: Atomic saturation.

An *atomic saturation* is a function (sat^A) that maps all interface nodes of a template graph to different interfaces of an atomic railroad net. The number of interface nodes of the graph must be equal to the number of interfaces of the railroad net. In that way we assure a one to one mapping between the nodes and the interfaces. Figure 6.1(a) shows a specification graph of a track segment and a railroad net representation of the same component. In Figure 6.1(b) the saturation is indicated by the dotted lines between the interface nodes of the specification graph, and the interfaces of the railroad net.

Definition 39 *Given a template specification graph* $\mathrm{G} = \langle \mathrm{N}, \mathrm{N}^{\mathrm{Int}}, \mathrm{L} \rangle$ *and an atomic railroad net* $\mathrm{R} = \langle \mathrm{CPN}, \mathrm{I} \rangle$*,* sat^A *is an atomic saturation of* G *with* R *if the following conditions are fulfilled:*

1. $\forall n \in \mathrm{N}\ \exists i \in \mathrm{I}$ *such that* $sat^A(n) = i$
2. $|\mathrm{N}| = |\mathrm{I}|$
3. $sat^A(n_1) = sat^A(n_2) \Rightarrow n_1 = n_2$

When saturating a specification language we must make an atomic saturation for each of the template components. When doing this one must be careful so that if two nodes are joinable, the corresponding interfaces must also be joinable. Figure 6.2 shows the template graphs of a specification language in the upper part of the figure, and a library of atomic railroad nets in lower part. The saturation is indicated by the dotted lines

Definition 40 *Given a specification language* $\mathrm{S}^{\mathrm{L}} = \langle \mathrm{G}^{\mathrm{T}}, \mathrm{D}, \mathrm{T}, \mathrm{C}, \mathrm{E} \rangle$ *and a library* $\mathscr{L}$ *of atomic railroad nets,* sat^L *is a library saturation function of* S^{L} *with* $\mathscr{L}$*, if:*

1. $\forall \mathrm{G} \in \mathrm{G}^{\mathrm{T}}\ \exists \mathrm{R} \in \mathscr{L} | sat^L$ *is an atomic saturation of* G *with* R
2. *If* n_1 *is an interface node in* G_1 *and* n_2 *is an interface node in* G_2*, and* $\mathrm{G}_1, \mathrm{G}_2 \in \mathrm{G}^{\mathrm{T}}$*. Then if* n_1 *and* n_2 *are joinable then* $sat^L(n_1)$ *and* $sat^L(n_2)$ *are joinable.*

For convenience we introduce a mapping function ($\mathcal{S}^{\mathcal{LG}}$) between template graphs and railroad nets. This function takes a library saturation function and a template graph as arguments. It returns the railroad net that the template graph is saturated with.

Definition 41 *Let* sat^L *be a language saturation function,* G *be a template graph, and* R *be an atomic railroad net. Then* $\mathcal{S}^{\mathcal{LG}}(sat^L, \mathrm{G}) = \mathrm{R}$ *if* sat^L *is an atomic saturation of* G *with* R

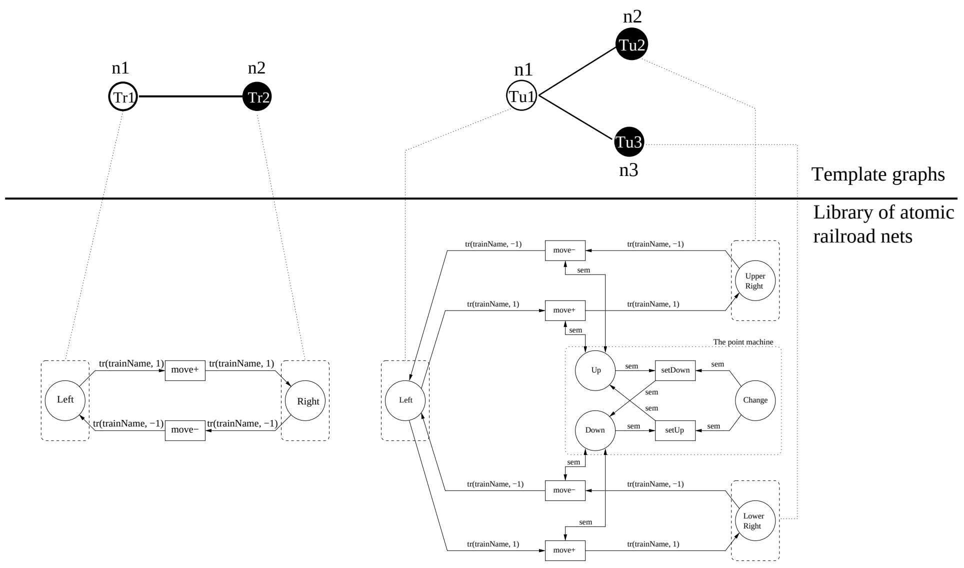


Figure 6.2: Saturation of a library

When saturating a specification all components of the specification graph is mapped to a copy of the railroad net that its template is mapped to. If two nodes in the specification are joined so are the corresponding interfaces of the underlying railroad nets.

Definition 42 *A saturation (sat) of a specification* $\mathrm{S} = \langle \mathrm{G}, \mathrm{S}^{\mathrm{L}} \rangle$ *with a library* $\mathscr{L}$ *by the language saturation function* sat^L *is defined by:*

1. $sat(\langle \mathrm{G}_1 \sqcap_{[n_1,n_2]} \mathrm{G}_2, \mathrm{S}^{\mathrm{L}} \rangle, \mathscr{L}, sat^L) =$
   $sat(\langle \mathrm{G}_1, \mathrm{S}^{\mathrm{L}} \rangle, \mathscr{L}, sat^L) \bowtie_{[sat^L(n_1), sat^L(n_2)]} sat(\langle \mathrm{G}_2, \mathrm{S}^{\mathrm{L}} \rangle, \mathscr{L}, sat^L)$

2. $sat(\langle \mathrm{G}, \langle \mathrm{G}^{\mathrm{T}}, \mathrm{D}, \mathrm{T}, \mathrm{C}, \mathrm{E} \rangle \rangle, \mathscr{L}, sat^L) =$
   $\mathscr{C}(\mathcal{S}^{\mathcal{LG}}(sat^L, \mathscr{T}(\mathrm{G})))$ *if* $\mathscr{T}(\mathrm{G}) \in \mathrm{G}^{\mathrm{T}}$

Equation 1 of the definition above states that if two interface nodes in the specification graph are joined, so are the corresponding interfaces of the underlying railroad net. Equation 2 expresses that a copy of template graphs is represented by a copy of the corresponding atomic railroad net. The saturation of a specification results in a executable railroad net implementation of the specification graph.

6.2 Colored Petri nets as Maude modules

There are several ways to translate a colored Petri net into a Maude module. We have focused on two approaches. They will be referred to as *set-marking* and *list-marking*. The names reflects the way markings are represented, either as a set or as a list in Maude. The two approaches will be analyzed and compared in Chapter 8.

6.2.1 Set-marking

Ölveczky [16] suggests a translation where all the tokens with their position are stored in a set. The declaration of the initial marking of the colored Petri net in Figure 6.3 will be:

```
eq initMarking = Left(train(+)) UpperRight(noTrain)
                 LowerRight(noTrain) Down(sem) .
```

The train names are omitted for readability reasons. For instance the term `Left(train(+))` reads "the place `Left`, contains the token `train(+)`", that is, `Left` contains a train token heading in the positive direction. The transitions

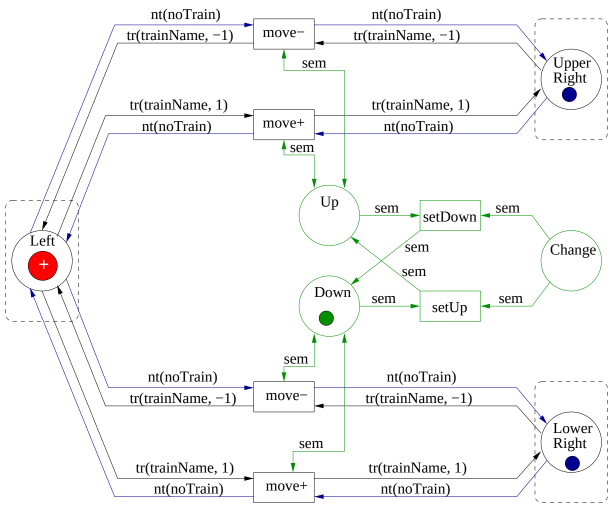


Figure 6.3: The safe turnout

will naturally be translated into rewrite rules. For instance the upper of the two `move-` transitions will become the rule:

```
rl [move-] : UpperRight(train(-)) Left(noTrain) Up(sem)
             =>
             Left(Train(-)) Up(sem) UpperRight(noTrain) .
```

The rule above is enabled if the current marking contains at least one occurrence of each of the three tokens in the left hand side of the rule. If the rule is executed, each of the tokens on the left hand side is replaced by the tokens on the right hand side of the rule. Since the marking is represented as a set, the order of the tokens in the rules are insignificant.

6.2.2 List-marking

In order to avoid extensive pattern matching we have proposed a translation where all places are stored in a list. All the places then contains a set of tokens. This could be a list, but because places i our systems would normally not contain many tokens, it really does not matter. The initial marking of the net in Figure 6.3 will be:

```
eq initMarking = [Left(train(+)), UpperRight(noTrain),
                  LowerRight(noTrain), Up(null), Down(sem),
                  Change(null)] .
```

All places will have a fixed position in the list. The `Left` place will always be first, `UpperRight` will be second and so on. This is done to minimize the number of pattern matches needed to find a token in a given place in a marking.

The rule for the upper `move-` transition is expressed in Maude as:

```
rl [move-] : [Left(noTrain), UpperRight(train(-)),
              lowerRight(toks), Up(sem), Down(toks'),
              Change(toks'')]
              =>
             [Left(train(-)), UpperRight(noTrain),
              lowerRight(toks), Up(sem), Down(toks'),
              Change(toks'')] .
```

where `toks`, `toks'` and `toks''` are variables of token set. Note that all the places are represented in the rule, not only the ones influenced by the transition, as is the case for set-marking. This is because of the fixed positions of the places in the list.

Chapter 7

Software

In this chapter we will give an overview of the software developed to build and execute railroad colored Petri nets. There will also be given a description of the process of setting up a simulation. The source code of the modules may be found on the attached CD-ROM.

7.1 The main tool

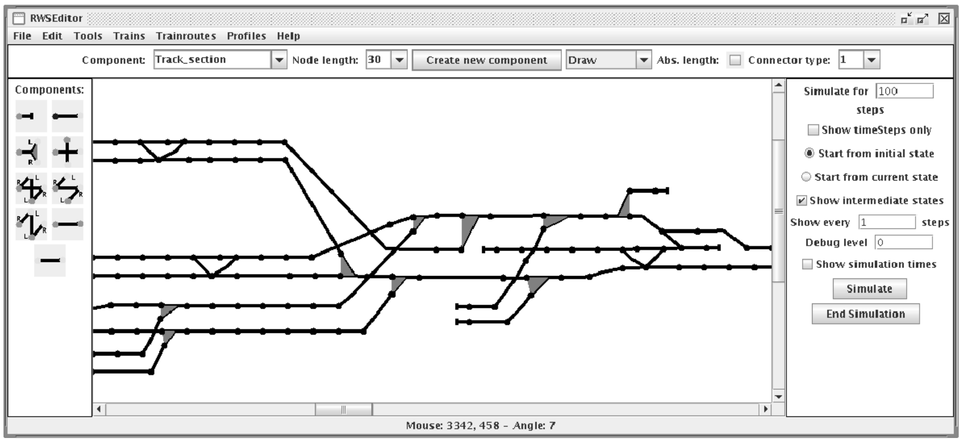
The main tool, RWSEditor, is a program designed to build up specification graphs from a predefined set of template graphs. It is also possible to assign types to the nodes of these template graphs. It has functionality for saturating the specification graph with imported atomic colored Petri nets. After the saturation it is possible to save the colored Petri nets as XML files. These nets are executable in Design/CPN, if their size is small enough. The original version of RWSEditor is written by Ingrid Chieh Yu, and is presented in [25].

The program has been expanded to handle interfaces. The atomic colored Petri nets are created in Design/CPN, and do not have any interfaces. These must be added in RWSEditor. The atomic colored Petri nets are thereby converted into railroad nets. Copies of these railroad nets are then connected by saturation of the specification graph. When saving the generated railroad net, the interfaces are omitted, again to obtain compatibility with Design/CPN.

In addition to assigning types to the nodes, one now has the possibility of assigning directions to the nodes of the template specification graphs.

Possibilities to create trains, train profiles and driver profiles have also been added. When the user wants to create a new profile, a form pops up. The fields of this form are dependent on the definition of the profile from the global declaration file. We have therefore implemented functionality for reading and analyzing declaration files.

Train routes may also be created by clicking on the nodes that are supposed to be in the route.



Component: Track_section
Node length: 30
Draw
Abs. length: ☐
Connector type: 1
Simulate for 100 steps
☐ Show timeSteps only
◉ Start from initial state
○ Start from current state
☑ Show intermediate states
Show every 1 steps
Debug level 0
☐ Show simulation times
Mouse: 3342, 458 - Angle: 7

Figure 7.1: Screen-shot of the RWSEditor

In order to be able to run online simulations trains must be displayed in the specification graph. We have also added functionality for communication with other programs such as Xml2Maude and rws-sim. Figure 7.1 shows a screen-shot of RWSEditor taken during a simulation. When simulating with RWSEditor one can choose how many rewrite steps one wants to simulate for, and also how many intermediate steps one wants displayed. For timed nets, one may also set how many time steps the simulation shall run.

7.2 The translation module

Xml2Maude is a translator made to translate railroad colored Petri nets to Maude modules. It handles both list- and set-markings as described in Chapter 6.2. In can either run as a separate program or be included in another program. When executed as a separate program the filename of the xml-file to be translated must be given in the command line. So has information on whether one want it to be translated by the set-marking or the list-marking approach. When called from another program the net can be transferred as a xml-element to prevent having to save it to a file.

This program is not a complete translator for all possible colored Petri nets constructed in Design/CPN. In order to be that it had to contain a complete compiler from ML to Maude, as the declarations in Design/CPN are written in ML. It is however a translator of a subset of these colored Petri nets sufficient for our purposes.

7.3 The automatic refinement tool

Autorefine is a Java program we have developed to automatically refine railroad colored Petri nets. It can refine single components, whole libraries or colored petri net representation of whole railway systems. When started, the program asks the user to select a refinement class from a menu. All the input files will be refined and written to a new file. Autorefine can refine the basic components discussed in Section 4.2 into all refinements described in this thesis.

7.4 The control script

The Perl script rws-sim controls the communication between RWSEditor and Maude. It is written by Pål Enger, and will be presented in his Cand. Scient. thesis.

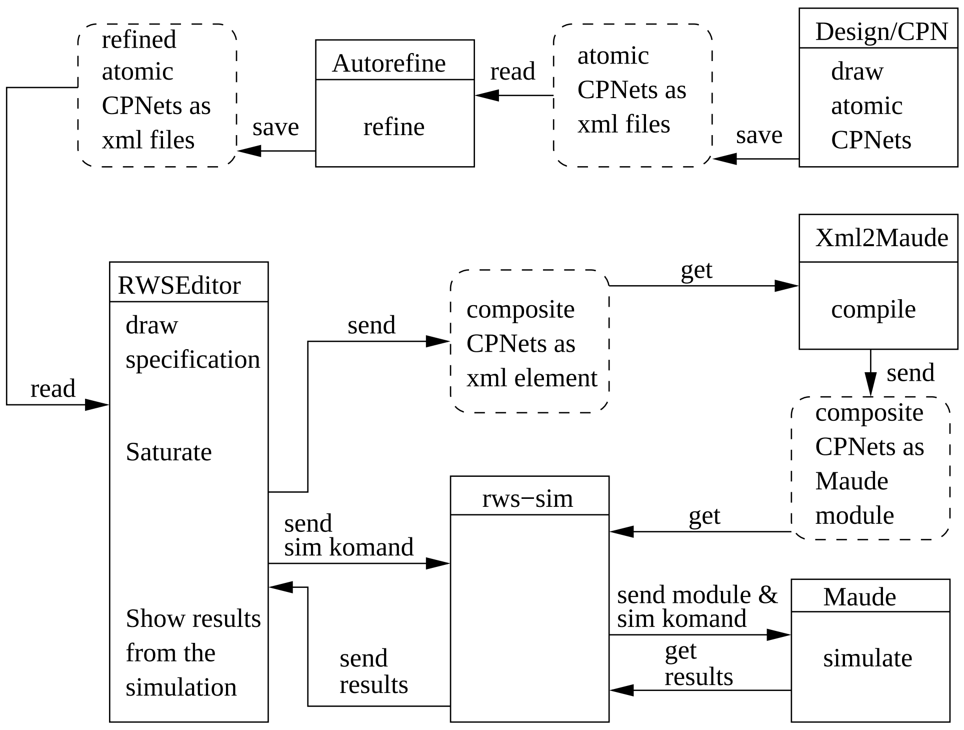


Figure 7.2: The overview of a simulation

7.5 How to set up a simulation

In this section a step by step description of the process of setting up a simulation is given. The system involves several modules as shown in Figure 7.2. Processes are drawn as rectangles. Events of a process are drawn in the order they occur, with the first event at the top. Files and data structures are drawn with dotted lines. The arrows represent the saving and loading of files, and sending and receiving data streams.

First the basic atomic colored Petri nets, such as the ones described in Section 4.2 are designed using Design/CPN. These nets are saved as xml-files. Then Autorefine is used to refine the basic atomic colored Petri nets to the desired level of refinement. The refined atomic colored Petri nets are also saved as xml-files following the same document type definition (dtd) as for the basic ones.

When all the refined atomic colored Petri nets are constructed, RWSEditor is started. In RWSEditor, one have to chose a set of template specification graphs. This should match the set of refined atomic colored Petri nets. Then multiple copies of these template specifications graphs are connected to form the specification graphs. The theory of this is described in Section 3.2. When the the specification graph is finished, it is time load the refined atomic colored Petri nets. Then interfaces should be added and the specification should be saturated with the railroad nets. This process is described in Chapter 6.1. Now would be a good time to add trains, train routes, profiles and so on. The system is now ready to be simulated. There are two ways of simulating the systems, *offline* or *online*.

We define offline simulation as a simulation where RWSEditor is used to save the colored Petri net, generated by a saturation, to a file. This file is translated to a Maude module by Xml2Maude and then executed in Maude. The output from an offline simulation is the textual output from Maude only. No train movements are shown in RWSEditor. With an online simulation the results of the simulation are displayed in the specification graph of RWSEditor. The first part is to display the initial positions of the trains on the specification graph. Then the whole system is converted to a xml-element. This xml-element is sent to Xml2Maude, which translates it to an executable Maude module. Then the simulation command is generated. This command contains information on how many steps the net should be executed, and whether every step should be displayed, or just some of them.

The control script, rws-sim, gets the Maude module and the simulation command, and starts the Maude engine. rws-sim then feeds Maude with the module and commands, and reports the results back to RWSEditor, which is used to displays the results.

Chapter 8

Simulation

In this chapter we will analyze the two different approaches of translating colored petri nets to Maude modules. We also describe some simulations, and present empiric data on their time consume.

8.1 Computational complexity

In this section we discuss the computational cost both with respect to time and space, and with respect to choosing either the set-marking or list-marking approach.

File size and computation cost of set-marking

The file-size of a colored Petri net CPN $= \langle P, T, A \rangle$ stored as a Maude module by using the set-marking approach will be given by a linear function of the expression:

$$|P| + |T| + |A| + |\text{tokens}|, \tag{8.1}$$

where |tokens| counts the number of tokens in the net.

In the basic railroad net components presented in section 4.2 all arcs leads one token. It is never more than one arc in each direction between two nodes. This also applies to all refinements of these nets used in this thesis. Hence a transition t consumes exactly $|{}^\bullet t|$ tokens when it fires.

In order to determine whether a given transition t is enabled, it is necessary to search through the tokens of the marking for each of the tokens in ${}^{\bullet}t$ to see if there is a matching rule. Thus as many as

$$|\text{tokens}| \times |{}^{\bullet}t| \tag{8.2}$$

tokens must be examined to decide whether t is enabled or not. In the colored Petri Net CPN, the number of tokens that must be checked in order to determine whether any transition is enabled, is given as the sum of expression (8.2) for all transitions $t \in T$:

$$\sum_{t \in T} |\text{tokens}| \times |{}^{\bullet}t| \tag{8.3}$$

Note that for every transition t, when two railroad nets are connected to each other $|{}^{\bullet}t|$ will not increase, hence $|{}^{\bullet}t|$ is independent of the size of the railroad.

File size and computation cost of list-marking

The file-size of a colored Petri net CPN $= \langle P, T, A\rangle$ stored as a Maude module by using the list-marking approach will be given by a linear function of the expression:

$$|P| + (|T| \times |P|) + |\text{tokens}|, \tag{8.4}$$

where $|\text{tokens}|$ counts the number of tokens in the net.

In order to determine if a transition t is enabled one have to find out if all places have a satisfying set of tokens. This means that it is necessary to compare the marking with the left-hand side of the rule. All places in the marking list is in the same position as in the list in the left-hand side of the rule, we therefore have to do

$$|P| \tag{8.5}$$

comparisons.

To be able to determine whether any transition is enabled we have to multiply expression 8.5 with the number of transitions in the system, and thereby get the expression:

$$|P| \times |T| \tag{8.6}$$

The sizes of the expressions 8.3 and 8.6, and the proportion between them are dependent on the net. Let us for instance consider colored Petri net representation of a large railway system, and use components with safety. Then the number of tokens are almost the same as the number of places. The number of input places of a transition is very small compared to these numbers. Therefore Expression 8.3 will be only a few times larger than Expression 8.6.

8.2 Numbers and statistics

Three sections of the Oslo subway system was modeled as specification graphs. The smaller one is the downtown part of the system, more precisely the tracks between the stations Majorstua and Grønland. This part is shown in Figure 2.5. In the second section, the downtown part was expanded westwards to the station Kolsås, and eastwards to Bergkrystallen. This is line number four of Oslo subway system. This specification graph was then expanded to a model of the whole Oslo subway system. The line map of the whole system is shown in Figure 2.4. These three specification graphs was saturated with several different refinements of the railroad nets presented in Section 4.2.

The three specification graphs were saturated with railroad nets with safety. This is the library presented in Section 5.1. Table 8.1 shows the size of the railroad nets constructed by the saturation. The table shows both the

Segment	Places	Transitions	Arcs	CPN file size	List-marking file size	Set-marking file size
Majorstuen - Grønland	242	423	2966	1.2 Mbyte	1.1 Mbyte	82 Kbyte
Kolsås - Bergkrystallen	885	1556	7710	4.3 Mbyte	14 Mbyte	311 Kbyte
The whole system	2067	3572	18286	10 Mbyte	79 Mbyte	728 Kbyte

Table 8.1: Size data of the simulated segments

number of railroad net components (places, transitions and arcs), and the file size of the nets stored as an XML-file, and as a Maude modules. As suspected the file size of the Maude module made as set-marking is much smaller than the one made as list-marking. The file sizes of the Maude modules relative to the Expression 8.1 for the set-marking and Expression 8.4 for the list-marking are almost the same for all three segments. These numbers are given in Table 8.2, and implies that the expressions are correct. The reason they are not exactly the same is probably that all constants are omitted in the expressions.

Segment	Expression 8.1 / set-marking file size	Expression 8.4 / set-marking file size
Majorstuen - Grønland	47	93
Kolsås - Bergkrystallen	35	98
The whole system	35	93

Table 8.2: Relative file size

First we simulated the three segments offline, as defined in Section 7.5. All three segments were saturated with safe components as presented in Section 5.1. Different number of trains were added to each of the segments, and then the colored Petri nets was saved. These colored Petri nets was translated to Maude modules with both the list-marking and the set-marking approach. Maude was then used to execute these modules. First the initial state was computed. Then one rewrite was executed, and finally all possible rewrites was performed. The simulation times as presented by Maude are shown in table 8.3.

As suspected the the simulation time increased as the net got larger. In the smaller net the simulation times were almost unmeasurable. This was the case for both list-marking and set-marking. In the net from Kolsås to Bergkrystallen, the set-marking was much faster than the list-marking. This is probably due to the large files of the modules represented by the list-marking approach. The list-marking of largest net was not executable on a normal modern computer due to its size, and are therefore omitted from the table. The set-marking version of the whole system however was executable in reasonable time. Because of this, the set-marking approach is used for the rest of the simulations presented in this thesis.

As suspected an increased number of trains results in an increased number of rewrites needed to reach a final state. However the simulation time did not increase. The reason is that the number of tokens in the system is not influenced by the number of trains. Each train token added to the net will replace a `noTrain` token. If the number of trains on the line is increased, more transitions are enabled during execution, which will decrease the time it takes to find an enabled transition.

segment	type	number of trains	Simulation time (milliseconds)			number of rewrites
			initial state	one rewrite	all rewrites	
Majorstuen - Grønland	set	1	0	0	30	85
		4	10	0	20	280
		16	0	0	20	953
	list	1	0	0	80	85
		4	0	0	70	280
		16	0	0	90	953
Kolsås - Bergkrystallen	set	1	0	0	900	1708
		4	0	10	910	11184
		16	10	0	920	32253
		32	0	0	980	39308
	list	1	1090	3680	15850	1708
		4	490	680	11200	4380
		16	260	790	19080	32065
		32	400	460	17110	39120
The whole system	set	1	0	80	4210	1614
		4	0	20	4070	5779
		8	0	30	4030	10100
		16	10	0	3960	37793
		32	0	10	3620	35630

Table 8.3: Experiments on off-line Simulation.

We also simulated the three segments online. We saturated all three specification graphs with the same atomic railroad nets as for the offline simulation. We ran the simulation with a various number of trains. The simulation times are presented in Table 8.4. First we simulated 100 steps and displayed the result of all of them in RWSEditor. Then we simulated 1000 steps, displaying every tenth step, Finally we ran the simulation until a final state was reached showing only the final state.

As for the offline simulation the simulation time per rewrite step decreased when the number of trains increased. None of the simulations for the smallest net were measurable, and are therefore omitted from the table.

We also ran simulations with railroad nets with constant delays, as presented in Section 5.3. We ran the simulation with a various number of trains and let it run for 50 time steps. The results of this simulation are presented in Table 8.5.

Execution time for the simulation of one step decreases with the number of trains, while the number of rewrite steps increases. The number of rewrite

Sample of lines	number of trains	Simulation time (milliseconds) 100 steps show all	1000 steps show every 10'th	all possible steps show only final
Kolsås	1	2879	1	0
-	10	2387	1	0
Bergkrystallen	30	1264	0	0
The complete	1	13435	1614	335
Oslo subway	10	13169	1337	40
system	30	12413	0	0

Table 8.4: On-line simulations with untimed nets.

Sample of lines	number of trains	number of time-steps	number of rewrite-step	average simulation time per time-step
The complete	1	50	98	141691 ms
Oslo subway	10	50	545	22042 ms
system	30	50	1528	66089 ms

Table 8.5: On-line simulations with timed nets.

steps is close to a linear function of the number of trains. Simulation will be much faster if we display fewer intermediate states.

Chapter 9

Conclusion

The three layered approach of representing railroad systems seems promising. The specifications are usable to communicate with a wide variety of people both railroad engineers and lay people. We are able to represent a wide variety of railroad systems. If the template graphs are not sufficient, new ones may be introduced.

The colored Petri nets (and the railroad nets) are suitable to model railroad components. When considered individually the colored Petri net components are fairly readable. Fortunately we do not have to look at the colored Petri nets representing the whole railroad systems. They are huge, and almost impossible to get a survey over.

The atomic colored Petri net components are refinable. This makes it possible to simulate different behaviors. In this thesis we have presented several different refinements. The simplest behavior we have looked at are nets where trains have no direction and just travel around by chance, These components are presented in [7]. We have introduced directions to the trains so that they do not turn around and run in the opposite direction every now and then these components are presented in Section 4.2 in this thesis. In [25] safety are introduced. This is a way of preventing two trains of passing each other on a single track. In Section 5.2 components where trains have travel plans are introduced. In these component train may also change the state of branching components. We have also looked at component with constant delays in Section 5.3. The most complex components considered so far are the components in Section 5.4. These components have a separate signal stream, and speed limits dependent on how far it is to the next obstacle. The speed of the trains are dependent on the profiles of the drivers and the trains. Further refinements are absolutely possible.

We have expanded the specification language by introducing directions. This makes it possible to create combination rules to prevent direction traps. The introduction of combination rules will arise some limitations. For instance, in the language defined in Section 3.3, one may not connect two turnouts by joining their stem ends. The direction converters have been introduced as a solution to these limitations. This is shown in Figure 3.15(a). We have also made it possible to create loops, as shown in Figure 3.4.

Railroad nets have been introduced as extensions to colored Petri nets. This makes it possible to group places into interfaces, Which have proven useful for the more complex components, such as the ones described in Section 5.4.

We have looked at two different approaches of translating colored Petri nets to Maude modules. These two approaches are referred to as the set-marking and the list-marking approach, and are presented in section 6.2. Both the analysis of the file-sizes of the Maude modules, and the execution times indicates that the set-marking approach is far better than the list-marking approach.

We have presented several software modules. The main program, RWSEditor, has been expanded in several ways. We have introduced a module for refining the colored Petri nets automatically, and another one for translating the colored Petri nets into executable Maude modules. Together with the control script, these programs form a working simulator.

The main problem of representing railroad systems as colored Petri nets is their size. The common tools for executing colored Petri net can not execute nets of this size, and would only be usable for simulation of tiny fragments of railroad system. We have looked at the possibilities of translating the colored Petri nets to Maude modules, and then use Maude to execute them. The results from the simulations shows that Maude can handle this large modules. We were able to simulate approximately one hour of real time traffic in the whole Oslo subway system in just a few seconds.

9.1 Related work

Our work is inspired by Wil van der Aalst's approach to Work-flow ([23], [24]) where Petri nets were used to precisely define, simulate, execute and analyze work-flows. The long-term goal of our research is to do the same for railway systems, decompose the application domain into its constituents and then show how Petri Nets can be used to enrich the software engineering of railway systems. A few papers devoted to work on modeling and analyzing railways using Petri Nets are available ([20], [8], [26], [3], [12]), but the papers are concerned with modeling and analysis of isolated phenomena. Two significant papers investigate the scheduling and analysis of movements on railway stations [21], and optimal behavior on the meeting points on single-track railways with sidings [19]. Safe control of train movements is investigated in [4]. A deadlock prevention method is introduced by augmenting railway network systems with monitor places. Our work differs from the other works in two respects: The models proposed in this paper are designed bottom up, and there is a close resemblance between the executable Petri nets and the actual railroad system.

9.2 Future work

There are several ways of continuing this work. We have divided these into three sections, even though some overlap occurs. A final goal may be to expand the system towards a product suitable for commercial use.

Further refinement

The are many ways of refining the atomic railroad net components, or even expand the set of basic atomic railroad nets. One could introduce a special station component. This component could delay the trains for a period of time dependent on the number of passengers getting on and off at this station. This could be a function of when the last train left the station, time of day etc.

A major task in refining the components is to implement the signal system. This may include both light signals and semaphores. This could probably be built upon the signal streams that is introduced in Section 5.4.

On may also consider other refinements. One of these could involve giving the trains length. All trains in all the refinements in this thesis are considered as points. One could also introduce possibilities for accidents, such as collisions.

Software development

The software could be expanded in many ways. One interesting option is to develop it into a tool for creation of time tables. In order to do so, the underlying railroad net must have a lifelike behavior. The tools could then be expanded to give access to the analyzing tools of Maude. The search functionality of Maude could for instance be used to search for the state where all trains have reached their destination in as short time as possible. The tool could then also be used for finding the best way of expanding a railroad system.

Another possible way is to make it into a training simulator for operators and dispatchers. This are the people who supervises and controls the train movements. One then need functionality of influencing the system during a simulation. This includes accessing the point machines and signals, and also giving orders to the train drivers.

Yet another possibility is to use the software to monitor trains at the control center. It then have to be connected to the hardware of the railway system.

Analysis

A more theoretical approach can be to analyze the nets. This can be done by using Maude's analyzing tools, or one could analyze small systems in a Petri net tool, such as Design/CPN. The analysis could involve searching for unwanted situations due to either badly designed railroad nets, or errors in the models.

Bibliography

[1] D. Bjørner and M. Pěnička. Towards a TRain book, 2004. Technical report, obtained from FMRail: `http://www.railwaydomain.org/`.

[2] M. Clavel, S. Eker, P. Lincoln, and J. Meseguer. Principles of Maude. Obtained from `http://maude.cs.uiuc.edu/papers/`, 1996.

[3] D. Decknatel. Modelling Train Movement with Hybrid Petri Nets. In *FMRail Workshop 4*, 1999.

[4] M. P. Fanti, A. Giua, and C. Seatzu. A Deadlock Prevention Method for Railway Networks using Monitors for Colored Petri Nets. In *Proc. 2003 IEEE Int. Conf. Systems, Man, and Cybernetics (Washington, D.C., USA)*, pages 1866–1873, October 2003.

[5] A. M. Hagalisletto, J. Bjørk, and P. Enger. Large scale simulations of railroad nets. Submitted to the 4th International Workshop on Modelling of Objects, Components, and Agents (MOCA 2006).

[6] A. M. Hagalisletto, J. Bjørk, I. C. Yu, and P. Enger. Constructing and Refining Large Scale Railroad Models Represented by Petri Nets, 2006. Accepted for publication IEEE Transactions on Systems, Man and Cybernetics, Part C.

[7] A. M. Hagalisletto and I. C. Yu. Large scale construction of railroad models from specifications. In W. Thissen, P. Wieringa, M. Pantic, and M. Ludema, editors, *IEEE SMC'2004. Conference Proceedings 2004 Systems Man and Cybernetics*, pages 6212 – 6219. IEEE, October 2004.

[8] W. Hielscher, L. Urbszat, C. Reinke, and W. Kluge. On Modelling Train Traffic in a Model Train System. In *Workshop and Tutorial on Practical Use of Coloured Petri Nets and Design/CPN, June 8-12, 1998, Aarhus Denmark*, 1998.

[9] K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*, volume 1 of *EATCS, Monographs on Theoretical Computer Science*. "Springer-Verlag", 1997. Basic Concepts.

[10] K. Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*, volume 2 of *EATCS, Monographs on Theoretical Computer Science*. "Springer-Verlag", 1997. Analysis Methods.

[11] T. Kristoffersen, A. M. Hagalisletto, and H. A. Hansen. Extracting High-Level Information from Petri Nets: A Railroad Case. *Proceedings of the Estonian Academy of Physics and Mathematics*, 52(4):378 – 393, December 2003.

[12] G. Malavasi and S. Ricci. Petri nets theory in the railway signalling models. In *FMRail Workshop 5*, 1999.

[13] A. M. Marsan, A. Bobbio, and S. Donatelli. Petri Nets in Performance Analysis: An Introduction. In W. Reisig and G. Rozenberg, editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 221–256. Springer-Verlag, 1998.

[14] N. Marti-Oliet and J. Meseguer. Rewriting Logic: Roadmap and Bibliography. Obtained from `http://maude.cs.uiuc.edu/papers/`, June 2001.

[15] J. Meseguer. Research directions in rewriting logic. In U. Berger and H. Schwichtenberg, editors, *Computational Logic, Proceedings of the NATO Advanced Study Institute on Computational Logic held in Marktoberdorf, Germany, July 29 – August 6, 1997*, volume 165 of *NATO ASI Series F: Computer and Systems Sciences*, pages 347–398. Springer-Verlag, 1998.

[16] P. C. Ölveczky. *Specification and Analysis of Real-Time and Hybrid Systems in Rewriting Logic.* PhD thesis, University of Bergen, 2000.

[17] J. Pachl. *Railway Operation and Control.* "VTD Rail Publishing", 2002.

[18] C. A. Petri. Kommunikation mit automaten. Technical Report Schriften des IIM Nr. 2, Bonn: Institut für Instrumentelle Mathematik, 1962.

[19] X. Ren and M. C. Zhou. Scheduling of Rail Operation: A Petri Net Approach. In *Proc. of 1995 IEEE Int. Conf. on Systems, Man, and Cybernetics, Vancouver, Canada, Vol. 4*, pages 3087–3092, October 1995.

[20] L. Tang, T. Chen, and J. Xiao. Analysis of the concurrent model of train station based on Petri net. In *Proc. 2000 Int. Workshop on Autonomous Decentralized Systems, 21-23 September 2000, Chengdu, China*, pages 92–96, 2000.

[21] W. van der Aalst and M. Odijk. Analysis of railway stations by means of interval timed coloured petri nets. *Real Time Systems*, 9(3):1–23, November 1995.

[22] W. M. van der Aalst. *Timed Coloured Petri Nets and their Application to Logistics.* PhD thesis, Eindhoven University of Technology, 1992.

[23] W. M. van der Aalst. Modelling and Analyzing Workflow using a Petrinet based Approach. In G. D. Michelis, C. Ellis, and G. Memmi, editors, *Proceedings of the second Workshop on Computer-Supported Co-operative Work, Petri nets and related formalisms*, pages 31–50, 1994.

[24] W. M. van der Aalst. The Application of Petri Nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–66, 1998.

[25] I. C. Yu. A Layered Approach to Automatic Construction of Large Scale Petri Nets. Master's thesis, University of Oslo, 2004.

[26] M. M. zu Hörste. Modelling and Simulation of Train Control Systems using Petri Nets. In *FMRail Workshop 3*, 1999.