University of Oslo
Department of Informatics

A Layered Approach to Automatic Construction of Large Scale Petri Nets

Modelling Railway Systems

Ingrid Chieh Yu

Cand. Scient. Thesis

25. august 2004

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 carried out at the research group *Precise modelling and analysis* (PMA).

I would like to thank everyone who helped me finish this thesis. First of all, I would like to thank my tutor on this thesis, Anders Moen, for his professional advices. Without whom there would not be any thesis.

I would also like to thank Einar Broch Johnsen for his proofreading and Thor Georg Sælid and Trygve Kaasa from Oslo Sporveier for the railroad layouts of Oslo subway and for sharing the knowledge about railroad engineering. Further I would like to thank Fredrik de Vibe for much technical input, discussions and not least for his patience. Finally thanks to my family and friends for their moral supports and encouragements.

Ingrid Chieh Yu
25 August 2004

Contents

List of Figures

Chapter 1

Introduction

Large concurrent systems, like distributed systems, are difficult to model and analyse, both conceptually and computationally. *Railway systems* are such large and complex concurrent systems. They are complex due to concurrent activity in railway components (e.g. trains), interaction between components (e.g. signals and track sections), many different behavioural possibilities and a variety of operational rules. These rules vary from system to system and in a given system, a combination of several operational rules may also be employed. Railway systems are also large in the sense that they often cover vast distances and contain many components: track arrangement, signalling equipment, locomotives etc. Railway systems are often under development. It is therefore necessary to be able to model and explore a system before it is built, in order to test different operational ideas and make presentations of systems we want to describe to other people.

Petri Nets [26] is a formal modelling language defined by Carl Adam Petri in his PhD thesis *"Kommunikation mit Automaten"* [23]. It is a generalisation of automata theory in which the concept of concurrently occurring events can be expressed. We believe Petri Nets provide a good framework for modelling railway systems. First, Petri Nets are very general and can be used to describe a large variety of different systems, on different abstraction levels, software and hardware, ranging from systems with much concurrency to systems with no concurrency. Second, Petri Nets have an explicit description of both states and actions, where actions represent changes from one state to another. Multiple actions may take place at the same time, giving a natural model of parallelism. Third, formal modelling languages have numerous advantages over informal languages, such as their precise meaning and the possibility to derive properties through formal proofs. Petri Nets support analysis by simulation and by more formal analysis methods. *Coloured Petri Nets* [17] are high level Petri Nets. They are based on original Petri Nets and have all the qualities described above, but they are extended with programming concepts. Coloured Petri Nets can provide primitives for definition of data types and manipulation of their values which is practical in industrial projects.

A challenge with Petri Nets is that when they grow, they tend to become hard to understand and work with[1]. This is specially true when it comes to industrial systems such as railway systems, as their complexities require large amounts of time in the modelling phase. In addition it is cumbersome to modify an existing net because of its complex structure. Even though hierarchical structures for Petri Nets have been investigated to some extent for some high level Petri Nets and [18] introduced techniques for extracting high level information from Petri Nets, until now there has been no effective techniques for constructing large Petri Nets.

The work on this thesis consists of three main parts:

1. Using Coloured Petri Nets to model railway systems with a component based approach, mostly focusing on the trackwork of the system and therefore disregarding signalling and control systems.
2. Defining a technique for automatic construction of large Petri Nets, in the domain of railway systems.
3. Implementing a tool using this technique.

The problems addressed in this thesis can be summarised by the following questions:

How can we use Coloured Petri Nets to model railway components
naturally with concrete operational rules and trains?

How can we automatically construct Petri Net models?
What kind of algebra is sufficient for this construction?
What are the benefits of this construction if any?

What are the benefits of analysis methods provided by
Petri Nets, when applied to railway systems?

---

[1] This is a general fact concerning most modelling and programming languages.

Design/CPN[1] is a computer tool supporting Coloured Petri Nets. The tool allows modelling, simulation and analysis of Coloured Petri Nets and is currently the most elaborate Petri Net tool. The current version of Design/CPN is distributed, supported and developed by the CPN group at the University of Aarhus, Denmark. Since developing a Petri Net simulator is not a part of this thesis, Design/CPN will be used for modelling, analysis and testing ideas.

Some of the subjects in this thesis are addressed in [34] and [21].

Overview

This thesis starts with presenting Petri Nets as the background material in Chapter 2, where we go through the most important Petri Net definitions. In Chapter 3 we will describe the basic railway components and show how these components can be modelled using Petri Nets. These components will be used when we construct railway topologies in later chapters. A problem regarding construction of large scale Petri Nets is addressed in Chapter 4, where we formally define techniques for automatic construction of Petri Nets, more specifically, directed to the construction of railway nets. In Chapter 5, we will demonstrate a tool based on these techniques along with some examples of its use. The demonstration of the tool carries on in Chapter 6, where it is used to construct a real life subway system. This application helps demonstrate the usefulness of the concepts described in Chapter 4 and such a tool. In Chapter 7, we will use the example given in Chapter 5 as the subject for further analysis, and finally, Chapter 8 presents the conclusion of this thesis and suggested further work.

Chapter 2

Petri Nets

Petri Nets was proposed by Carl Adam Petri in his PhD of 1962 as a mathematical notion for modelling distributed systems and, in particular, notions of concurrency, non-determinism, communication and synchronisation [23]. Since then Petri Nets has been developed tremendously in both theory [7] and application [25]. One of the main attractions of Petri Nets is the way in which the basic aspects of concurrent systems are identified, both conceptually and mathematically. As a graphically oriented language Petri Nets eases conceptual modelling and makes it the model of choice for many applications.

2.1 Informal Introduction

Petri Nets have few basic primitives [26]. A Petri Net is a directed graph with nodes that are called *places* or *transitions*. Places are drawn as circles describing the states of the system being modelled. Places may contain *tokens*, and at any time the distribution of tokens among places defines the current state of the modelled system. This distribution of tokens in a Petri Net is called a *marking*. Transitions describe actions and are drawn as rectangles. They model activities that can occur (transitions can fire) thus changing the state of the system. Nodes of different kinds are connected by *arcs*. There are two kinds of arcs, *input arcs* and *output arcs*, an input arc connects a place to a transition and an output arc connects a transition to a place.

Change of states occurs by *transition firings*. Transitions are only allowed to fire if they are *enabled*, which they are when each of a transition's input places contain at least one token. This means that all the preconditions for an activity must be fulfilled before the state changes. When a transition is enabled, it may *fire*, removing a token from each of its input places and depositing a token in each of its output places, corresponding to the postcondition of the activity. The number of input and output tokens may differ, and the interactive firing of transitions in subsequent markings is called a *token game*.

Figure 2.1 shows a Petri Net in the simplest form, illustrating the firing of a transition t. The transition is enabled since all its input places contain a token (before firing). Firing t results in removing token from each input place of t and adding one token to each of its output places (after firing).

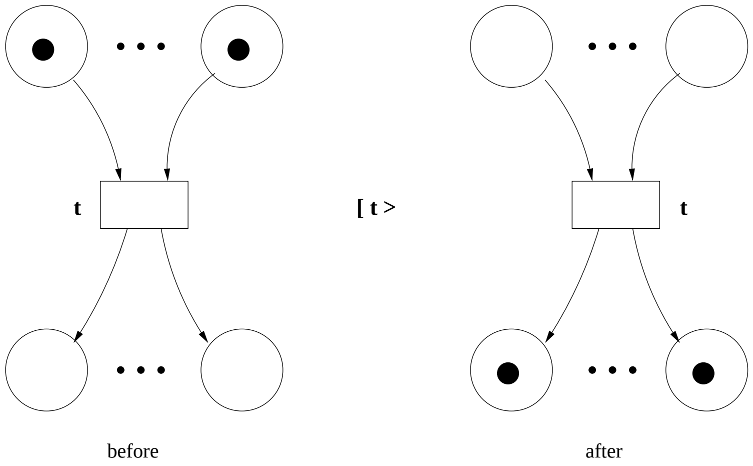


Figure 2.1: Firing of a transition

A Petri Net model consists of a net and the rules of a token game played on the net. The net describes the static structure of a concurrent system and the rules describe its dynamic behaviour. Many different classes of Petri Nets have been developed and they differ in the construction of the underlying net and the rules of the dynamic token game [24]. Therefore, Petri Nets is actually a generic name for the whole class of net-based models which can be divided into three main layers:

Level 1:
Petri Nets characterised by places that can represent boolean values, i.e., a place is marked by at most one unstructured token.

Level 2:
Petri Nets characterised by places that can represent integer values, i.e., a place is marked by a number of unstructured tokens.

Level 3:
Petri Nets characterised by places that can represent high-level values, i.e., a place is marked by a multi-set of structured tokens.

Petri Nets at the first two levels are not well suited for modelling large real-life applications. This is due to the fact that Petri Nets in the first two levels have no data concepts since tokens are unstructured. Hence the models often become excessively large as all data manipulations have to be represented directly in the net structure in the form of places and transitions. In addition, there are no hierarchy concepts, thus it is not possible to build a large model via a set of separate sub-models with well-defined interfaces.

In this thesis, work is primarily based on high-level Petri Nets, in particular, *Coloured Petri Nets* [12] (also called CPN or CP-nets). Coloured Petri Nets combine the strengths of ordinary Petri Nets with the strengths of a high-level programming language. Petri Nets provide the primitives for process interaction, while the programming language provides the primitives for the definition of data types and the manipulations of data values, making Coloured Petri Nets suited to model real-life applications. The most important definitions of Coloured Petri Nets will be given.

2.2 Coloured Petri Nets

Coloured Petri Nets [16, 15] got their name because they allow the use of tokens that carry data values and can hence be distinguished from each other, in contrast to tokens of low level Petri Nets, which by convention are drawn as black dots.

In this section, we will present the necessary definitions of Coloured Petri Nets from [13], introducing their structure before considering their behaviour.

Formally, a Coloured Petri Net is defined as follows:

Definition 1 ***Coloured Petri Net***
A Coloured Petri Net is a tuple $CPN = (\Sigma, P, T, A, N, C, G, E, I)$ *where:*

1. Σ *is a finite set of non-empty types, called colour sets.*
2. P *is a finite set of places.*
3. T *is a finite set of transitions.*
4. A *is a finite set of arcs.*
5. N *is a node function connecting places and transitions.*
6. C *is a colour function.* $C : P \longmapsto \Sigma$.
7. G *is a guard function.* $G : T \longmapsto Expr$.
8. E *is an arc expression function.* $E : A \longmapsto Expr$.
9. I *is an initialisation function.* $P \longmapsto closedExpr$.

An example Coloured Petri Net is provided in Section 2.2.8. We shall now look closer at the different components.

2.2.1 Colour Sets

A Coloured Petri Net has *colour sets* (= types). The set of types determines the data values and the operations and functions that can be used in the net expressions. A type can be arbitrarily complex, defined by means of many sorted algebra as in the theory of abstract data types. Examples of types are Integers, Boolean values, Strings, and more complex types such as Tuples, Products, Lists, etc.

Each place in a Coloured Petri Net has an associated colour set that determines what kind of data the place can contain. For a given place, all tokens must have data values that belong to the type associated with the place. The colour function C maps each place p to a type $C(p)$, formally defined from P into Σ. This means that each token on p must have a data value that belongs to $C(p)$.

2.2.2 Guards

Transitions in a Coloured Petri Net may also have *guards*. Guards are boolean expressions that provide additional constraints that must be fulfilled before transitions can be enabled. We denote the type of a variable v by *Type(v)*, the type of an expression *expr* by *Type(expr)*, and the set of variables in an expression by *Var(expr)*. Types of variables in a guard expression must belong to the set of colour sets. Formally, a guard must satisfy the following condition:

$$\forall t \in T : [Type(G(t)) = Boolean \wedge Type(Var(G(t))) \subseteq \Sigma]$$

In Coloured Petri Nets, guard expressions that always evaluate to true are omitted.

2.2.3 Arc Expressions

Before we describe arc expressions in Coloured Petri Nets, we must first define *multi-sets*. This is because tokens in a Coloured Petri Net may have identical token values and arc expressions evaluate to multi-sets of tokens. A multi-set may contain more than one occurrence of the same element.

Definition 2 ***Multi-sets***
A multi-set m, over a non-empty set S, is a function $m \in [S \to \mathbb{N}]$ which we represent as a sum:

$$\sum_{s \in S} m(s)'s$$

The non-negative integers $m(s) \in S$ are the coefficients of the multi-set, the number of occurrence of the element s in the multi-set m and $s \in m$ iff $m(s) \neq 0$.

Given a set S and $s \in S$, we use $m(s)'s$ to denote that element s occurs $m(s)$ times in the set S. If $C(p)$ is the type of a place p then $C(p)_{MS}$ denotes the multi-set over the type $C(p)$.

Arcs may have arc expressions that describe how the state of the CP-net changes when transitions fire. The arc expression function E maps each arc a into an expression of type $C(p)_{MS}$, which is a multi-set over the type of its place p. The variables in each arc expression must also form a subset of the colour sets. Formally, this means:

$$\forall a \in A : [Type(E(a)) = C(p)_{MS} \wedge Type(Var(E(a))) \subseteq \Sigma]$$

Having defined the structure of Coloured Petri Nets, their behaviour may now be considered, but it is first necessary to define the binding of variables, tokens and states in a Coloured Petri Net.

2.2.4 Bindings, Tokens and Markings

For a transition to occur, its variables must be bound to values of their types. The variables of a transition t are variables that occur in its guard expression and in its input and output arcs expressions. Formally, this is denoted by the set:

$$Var(t) = \{v | v \in Var(G(t)) \vee \exists a \in A(t) : v \in Var(E(a))\}$$

where $A(t)$ gives all input and output arcs of t.

The binding of a transition t is then a function b defined on $Var(t)$.

Definition 3 ***Binding of a transition***
A binding of a transition t is a function b defined on Var(t), such that the following equation evaluates to true:

$$\forall v \in Var(t) : b(v) \in Type(v) \wedge G(t)\langle b \rangle$$

The set of all bindings for t is denoted by $B(t)$.

$G(t)\langle b \rangle$ denotes the evaluation of the guard expression $G(t)$ in the binding b.

Definition 4 *A token element is a pair* $\langle p, c \rangle$ *where* $p \in P$ *and* $c \in C(p)$. *A binding element is a pair* $\langle t, b \rangle$, *such that* $t \in T$ *and* $b \in B(t)$

TE denotes the set of all token elements.
BE denotes the set of all binding elements.

Now we may define markings of a Coloured Petri Net. A marking consists of a number of tokens positioned in the individual places and describes a state of a Coloured Petri Net.

Definition 5 *A marking M is a multi-set over TE. The initial marking M_0 is the marking which is obtained by evaluating the initialisation function I:*

$$\forall \langle p, c \rangle \in TE : M_0(\langle p, c \rangle) = (I(p))(c)$$

A Marking is often represented as a function defined on P, and returns a multi-set of tokens. If M is a marking and p a place, we denote by $M(p)$ the number of tokens in p in the marking M. The initialisation function I maps each place p into a closed expression that must be of type $C(p)_{MS}$. The initial marking describes the initial state of a Coloured Petri Net.

2.2.5 Enabling and Firing

The dynamic behaviour of Coloured Petri Nets is provided by firing of transitions, and a transition can only fire when it is enabled. This behaviour is also non-deterministic, for example, if multiple transitions are enabled at the same time, multiple transitions may fire in one step, but the number is non-deterministic.

A transition t is enabled if a *step* is enabled with t. A step is a multi-set over the set of binding elements BE. Let $E(p, t)$ denote the arc expression of an arc from place p to transition t and let $E(t, p)$ denote the arc expression of an arc from transition t to place p. Enabling and firing of steps are always related to the current marking of the net.

Definition 6 *A step Y is enabled in a marking M if and only if:*

$$\forall p \in P : (\Sigma_{(t,b) \in Y} \; E(p, t)\langle b \rangle \leqslant M(p))$$

The expression $E(p, t)\langle b \rangle$ gives the number of tokens required from each place p to enable t and t is enabled if and only if each p contains at least as many tokens ($M(p)$). When $|Y| \geqslant 1$, elements of Y are concurrently enabled.

When a transition is enabled with a given binding it is ready to fire. Firing a transition removes at least one token with proper value from each of its input places and deposits at least one token in each of its output places. For a concrete transition t, firing t with binding b means that for each place p, $E(p, t)\langle b \rangle$ number of tokens are removed from p and $E(t, p)\langle b \rangle$ tokens are given to p.

Definition 7 *Let Y be a step that is enabled in a marking M_1. Then Y might fire from M_1 to M_2:*

$\forall p \in P, M_2(p) = (M_1(p) - \Sigma_{\langle t,b\rangle \in Y} E(p,t)\langle b\rangle) + \Sigma_{\langle t,b\rangle \in Y} E(t,p)\langle b\rangle$

"M_2 is reachable from M_1 in one step" is noted as $M_1 \xrightarrow{Y} M_2$

By taking the sum over the multi-sets of binding elements $(t,b) \in Y$, we get all the tokens that are removed from p when Y occurs. This multi-set is required to be less than or equal to the marking of p, meaning that each binding element $(t,b) \in Y$ must be able to get the tokens specified by $E(p,t)\langle b\rangle$ without sharing these tokens with other binding elements of Y. As for non-determinism, when a number of binding elements are enabled at the same time, there can be a possible step that only contains some of them or if two binding elements (t_1,b) and (t_2,b) share tokens specified by $E(p,t_1)\langle b\rangle$ and $E(p,t_2)\langle b\rangle$ then it is non-deterministic which one of them will fire, either $(t_1,b) \in Y$ or $(t_2,b) \in Y$.

A step is an indivisible event, even in the definition of firing of a step the subtraction is performed before the addition. The continuing firing of steps from one marking to the next may be finite or infinite. The finite firing sequence of markings and steps is:

$$M_i \xrightarrow{Y_i} M_{i+1} \ \forall i \in \{1, 2, \ldots, n\},$$

while the infinite firing sequence continues forever:

$$M_i \xrightarrow{Y_i} M_{i+1} \ \forall i \in N$$

If a marking M_j is reachable from a marking M_i then there exists a finite firing sequence from M_i to M_j and is written

$$M_i \xrightarrow{*} M_j$$

where $*$ means zero or more steps. We denote the set of markings reachable from a marking M by M^R and a marking is reachable if and only if it belongs to the set of markings reachable from the initial marking, M_0^R.

2.2.6 Declarations

Design/CPN uses the language CPN ML [19] which is an extension of Standard ML. Colour sets are declared with **color**.

Integers

Integers are numerals without a decimal point and can be restricted by the **with** clause.

color colourset_name = **int** « **with** int-exp$_{\text{start}}$ ... int-exp$_{\text{end}}$ »;

int-exp$_{\text{start}}$ and int-exp$_{\text{end}}$ restrict the integer colourset to an interval and the expression int-exp$_{\text{start}}$ must be equal or less than int-exp$_{\text{end}}$.

Enumerated values

Enumerated values are explicitly named as identifiers in the declaration and must be alphanumeric.

color colourset_name = **with** id$_0$ | id$_1$ | ... | id$_n$;

Tuples

Tuples are compound colour-sets. The set of values in a tuple is identical to the cartesian product of the values in previously declared colour-sets. Each of these colour-sets may be of a different kind and there must be at least two colour-sets to form a tuple.

color colourset_name = **product** colourset_name$_1$ * colourset_name$_2$ * ... * colourset_name$_n$;

Lists

Another compound colour-set is a *list*. Lists are variable-length colour-sets unlike tuples, which are fixed-length and positional colour-sets. In lists, the values are a sequence whose colour-set must be the same type.

color colourset_name = **list** colourset_name$_0$ « **with** int-exp$_{\text{b}}$ ... int-exp$_{\text{t}}$ »;

The minimum and maximum length of the list can optionally be specified by the **with** clause.

List operators and functions are the same as in Standard ML. The prepend operator, ::, creates a list from an element and a list by placing the element at the head of the list. The concatenation operator is denoted by ^^, unlike @ in Standard ML. This is because @ is used in Coloured Petri Nets to denote time. The concatenation operator takes two lists and appends one list to the other.

Union

Colour-set *union* is a union of previously declared colour-sets.

color colourset_name = **union** id_1 «:colourset_name_1» + id_2 «:colourset_name_2» + ... + id_n «:colourset_name_n»;

Each id_i is a unique selector for colour-set_i. If colourset_name_i is omitted, then id_i is treated as a new value and may be referred to as id_i.

2.2.7 Notations

Most notation used in Coloured Petri Nets is also used in Design/CPN.

Each place and transition has a name written inside respectively the circles and squares.

Types are written in *Italic* letters over each place, and each token is represented as a coloured circle inside a place. If this notation for tokens is used, then an explanation of their types is given in a colour map. In the syntax of Design/CPN, tokens are written as strings on the form n‘s, representing a multi-set where n is the number of tokens of type s. The addition of different types is represented by ++.

A guard expression is written in brackets and located next to its transition. Each arc expression is located next to its arc. The coefficients of the multi-set of an arc expression is omitted if it is 1.

2.2.8 An Example

The model in Figure 2.2 on the next page is a modification of an example from [20]. It shows a train station ticket office where train passengers buy tickets before they enter the platform. If the passenger is a child, he needs a child’s ticket, if the passenger is an adult, he needs an adult’s ticket, in order to enter the platform. A clerk sells the tickets and he obtains the correct tickets from a ticket machine.

There are four places, one transition and six arcs, each place has a colour set. These four colour sets are; *Buyer, Passenger, Staff* and *Ticket.*

color Buyer = with Child / Adult;

color Passenger = Buyer;

color Staff = with Clerk;

color Ticket = with ChildTicket / AdultTicket;

var buyer: Buyer;

var staff: Staff;

var ticket: Ticket;

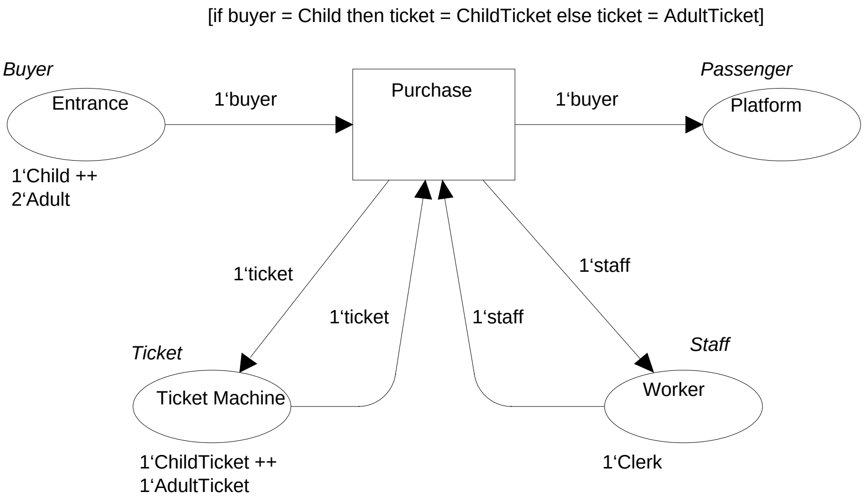


Figure 2.2: CPN model of a simple sales order processing system

As indicated by the string representation of the multi-sets, the place *Entrance* contains three tokens; one of value *Child* and two of value *Adult*, the place *Worker* contains one token of value *Clerk*, the place *Ticket Machine* contains two tokens; one of value *ChildTicket* and one of value *AdultTicket* and the place *Platform* has no tokens. The markings in all these places constitute the current state of the net.

In this example, in order for the transition *Purchase* to fire, there must be:

- At least one *Buyer* waiting in the *Entrance* to the ticket office.
- At least one *Staff* member who is working.
- At least one *Ticket* of the appropriate type in the. *Ticket Machine*

One of the possible *bindings* in this example is

$b = \langle \mathit{buyer} = \mathit{Child},\ \mathit{staff} = \mathit{Clerk},\ \mathit{ticket} = \mathit{ChildTicket} \rangle$.

With this binding, the guard expression above the transition will be evaluated to *true* and transition *Purchase* is enabled. Output arc expressions specify that firing *Purchase* will put a *Staff* token into *Worker*, a *Ticket* token into *Ticket Machine* and a *Passenger* token into *Platform.* This binding represents a situation where a child buyer has bought a child's ticket and enters the train platform. The clerk is ready to serve another buyer and the ticket machine generates a new child's ticket.

2.2.9 Dynamic Properties

Dynamic properties characterise the behaviour of Coloured Petri Nets. Some of the most interesting questions we would like to have answered are:

- Is a given marking reachable from the initial marking?
- Is it possible to reach a marking in which no transition is enabled?
- Is there a reachable marking that puts a token in a given place?

Most problems can be categorised as *boundedness* or *reachability* problems.

Boundedness properties

Boundedness properties tell how many tokens a particular place may contain.

Definition 8 *Given a place* $p \in P$*, a non-negative integer* $n \in \mathbb{N}$ *and a multi-set* $m \in C(p)_{MS}$*. Then* *n is an integer bound for p iff*

$$\forall M \in M_0^R : |M(p)| \leqslant n$$

m is a multi-set bound for p iff

$$\forall M \in M_0^R : M(p) \leqslant m$$

Upper integer bounds give the maximum number of tokens each individual place may have and *lower integer bounds* give the minimum number of tokens. An *upper multi-set bound* of a place is the smallest multi-set which is larger than all reachable markings of a place. Analogously, the *lower multi-set bound* is the largest multi-set which is smaller than all reachable markings of the place. The integer bounds give information about the number of tokens while the multi-set bounds give information about the values the tokens may carry.

Liveness Properties

Liveness properties are about reachability, whether a set of binding elements X remains active such that it is possible for each reachable marking M to find an occurrence sequence starting in M and containing an element from X. Some of the most interesting liveness property are deadlock and progression.

If M is a marking in which no transitions are enabled, then M is called a *dead marking. Dead transitions* are transitions that never are enabled. In contrast, a *live transition* is a transition that always can become enabled once more. This means that, if a system has a live transition, there cannot be any dead markings in that system.

Definition 9 *Let M be a marking and $Z \subseteq BE$ be a set of binding elements, then:*

- *M is dead iff no binding is enabled in M.*
- *Z is dead iff no binding elements of Z can become enabled.*
- *Z is live iff there is no reachable marking in which Z is dead.*

2.3 Analysis Methods

A Coloured Petri Net can be analysed by means of simulations and by formal methods such as state space analysis[1][14].

Formal methods can be used to verify that a formal system has a stated property, analyse the system or detect errors. For a railway system we may use formal methods to verify for example essential safety questions or questions regarding performance of trains.

2.3.1 Simulation

A Coloured Petri Net may be simulated manually or using a computer tool. Simulation can never give proof of correctness of a system but only reveal errors. A simulation run gives us one possible behaviour of the modelled system with details of each step. During a simulation, it is possible to watch all occurring transitions, input tokens, output tokens and markings.

In Design/CPN, the occurrence of enabling transition may be adjusted. It is possible to force some or all enabled transitions in a marking to fire in one step.

2.3.2 State Space Analysis

State space analysis (also called occurrence graphs or reachability graphs) is often complemented by simulations. The basic idea underlying state spaces is to compute all reachable states, all possible occurrence sequences and state changes of the system, and represent these as a directed graph, called occurrence graph. Each reachable marking is represented as a node, and nodes are connected by arcs. An arc represents an occurring binding element that changes its predecessor marking to its successor marking.

Calculating the occurrence graph of a Petri Net may give a lot of useful information about the behaviour of the net. The analysis method is based on answering queries about the dynamic behaviours of the net by performing searches through the occurrence graph.

---

[1] There are many other formal analysis methods like reductions, calculation and interpretation of system invariants and checking of structural properties.

Chapter 3

Mapping Railroad Components to Petri Nets

In the process of modelling railway systems using Coloured Petri Nets, it is natural to consider how railway components and operations may be represented as realistic as possible. It is desirable to model railway components in such a way that they can be reused multiple times in a railway network to form different topologies. This requires basic railway components to be modelled with respect to the following three properties:

1. Modularity, independence of others.
2. Topology independence.
3. Dynamic behaviour represented by tokens.

Modelling railway components as Petri Net modules allow us to construct large Nets by compositions of the different Petri Net railway components. This way of constructing railway nets reflects how railroads are constructed in real life. These modules must be modelled in such a way that they can be used to form the topologies we need, hence they must be topology independent. Properties that are considered to be topology independent are e.g. safe train separation and the logic of railroad components. These topology independent properties can be included in the Petri Net components as structures, while topology dependent properties and the dynamic behaviour is represented by tokens.

This chapter is dedicated to describing how different railroad components can be modelled in Coloured Petri Nets, how some operational rules are built into these components and which auxiliary functions we need in relation to the operational semantic. These components are the basis for construction of railway topologies and analysis in later chapters. Before we consider modelling of railway components in Coloured Petri Nets, an overview of the basic railway components is given in Section 3.1.

3.1 Railway Systems

A railway system consists mainly of three essential elements [22]. The first is the infrastructure with trackwork, signalling equipment and stations. The second is rolling stock with cars and locomotives and the third element is different operating rules and procedures for a safe and efficient operation. In this thesis, the focus is on the trackwork, trains and the primary operating rules for safe train separation.

3.1.1 Basic Components

In railway systems, there are many different ways to arrange rails that create different topologies. Even though there are many different topologies, with varying complexity, we may classify elements in these topologies, and a railway network can then be seen as a way to assemble these elements. These elements are therefore components of railway systems.

Some basic components that we consider are track segments, end segments, turnouts, double slips, rigid crossings, scissors and singles. These components are described and further explained as follows.

End Segment and Track Segment

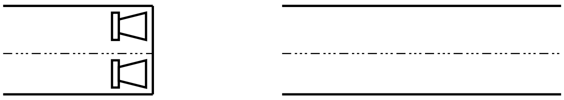


Figure 3.1: A railroad end and track segment

The *track segment* is the main building block for constructing railroads and models physical extension of a line.

Turnout

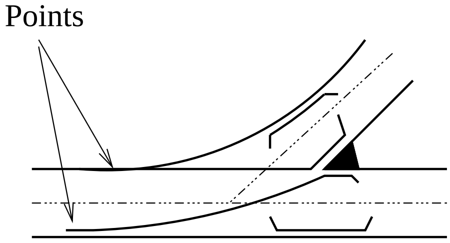


Figure 3.2: A railroad turnout

A *turnout* (Figure 3.2) is an assembly of rails and movable physical points. The points are operated electrically by a point machine to alter the direction in which trains are routed. The turnout permits the trains to be routed over one of two tracks, depending on the position of the points. In addition to be the name of a railroad component, we use the word “turnout” to describe the junctions in trackwork where lines diverge or converge, and, as we will see, there are a number of components that uses the concept of turnouts.

Double Slip

A *double slip* (Figure 3.3) is a crossing with crossover on both sides. It has two point machines such that at each entry of the double slip, trains may be routed to one of two tracks. This component is appropriate to use when the area is too narrow for a scissors crossing (Figure 3.5).

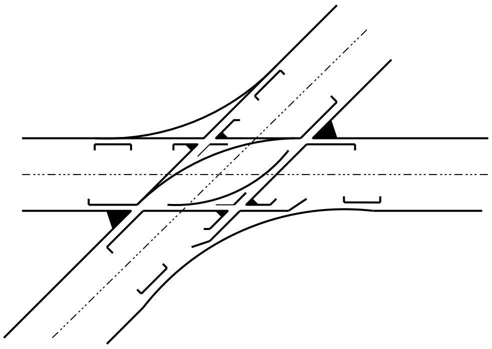


Figure 3.3: A railroad double slip

Rigid Crossing

A *rigid crossing* (Figure 3.4) effects two tracks to cross at grade. It is a crossing without movable points.

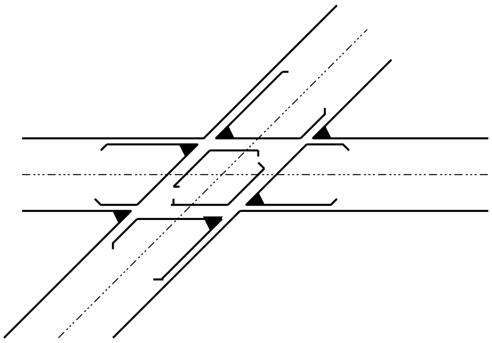


Figure 3.4: A railroad rigid crossing

Scissors

A *scissor* (Figure 3.5) is a track structure that connects two parallel track with an X-shaped crossover. It consists of 4 turnouts and 1 rigid crossing.

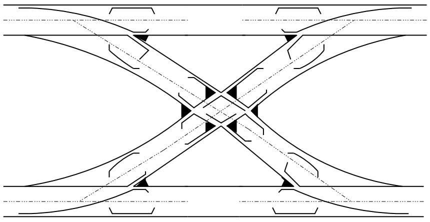


Figure 3.5: A railroad scissor

Single

A *single* (Figure 3.6) provides a connection between two parallel tracks. Two singles can be combined to construct a *universal*, which is a structure that allows trains moving in both directions to cross over to the adjacent track.

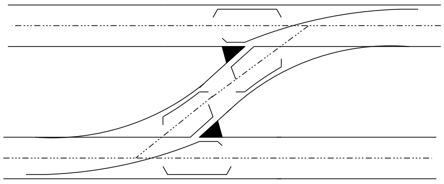


Figure 3.6: A railroad single

Both scissors and singles are railroad constructions that are commonly used in today's railroad designs.

3.2 Safety Components

Safety components are components that provide a safe train operation where all trains are separated at any time, making collisions impossible. In [18], non-safe road and turnout components were introduced, allowing trains to pass each other on a physical line. In this section, it will be shown how these components can be modified into safety components.

For simplicity and readability, some arc inscriptions from places with singleton types are omitted. Empty arc inscriptions denote tokens with data structures equal the corresponding places.

3.2.1 Trains and Road Components

Trains are tokens with data structures:

*color Train = product TrainLineNo * Direction;*

where

color TrainLineNo = int;

color Direction = with CL | ACL;

TrainLineNo represents the train lines and *Direction* represents the two directions each train line may have, either clockwise *CL* or anti-clockwise *ACL*. The terms clockwise and anti-clockwise have nothing to do with any curvature of train lines, they are simply names of the two possible directions in which a train may move.

Let n and dir be variables respectively of type *TrainLineNo* and *Direction*, then the variable $tr(n, dir)$ represents a train with its corresponding attributes. To distinguish trains with identical routes, we may give each train a unique identity.

An important concept in railway systems is the *block system*. A block system defines how to divide lines into fixed block sections to provide a safe train separation by ensuring that at most one train can be in any section at any time. In an *automatic block system* the clearance of block sections is done by *track clear detection devices*, which are device that detects whether a track section is occupied.

Figure 3.7 represents a road component focusing on the basal elements of the block system. It consists of two *segment places* representing physical track sections and two *move transitions*, one for each direction, for moving trains from one segment place to the other. The railroad track section modelled by segment places in a Petri Net road component is coherent, making it possible to drive a train over it. This is equivalent to a token moving from one segment place to the next.

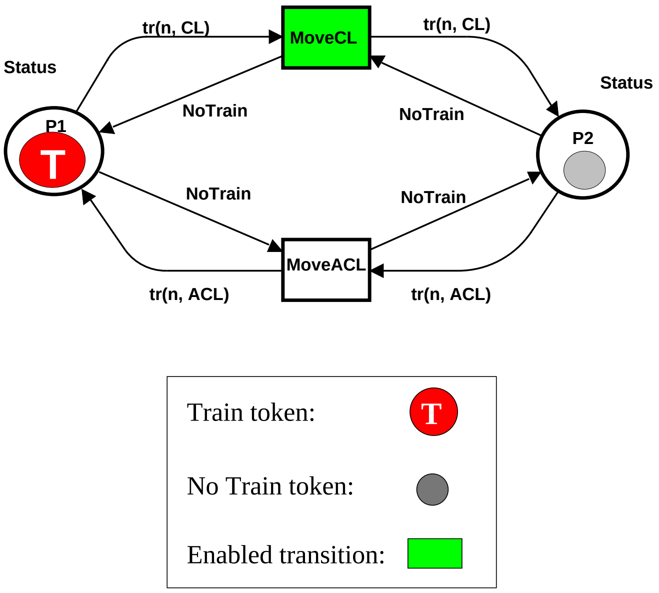


Figure 3.7: A road component

Each segment place has the type *Status*:

color Status = union tr:Train + NoTrain;

Tokens of this type represent the absence or presence of trains in a section. As shown by the arcs, which implement the theory behind track clear detection, a *move transition* is enabled if a train is in section *P1 (P2)* and no train is in section *P2 (P1).* The transition can then fire, exchanging two tokens, simulating that the train moves on. For controlling train movement from one section to the next, there must be a token of either type residing in each segment place under all markings of the net. In the component in Figure 3.7, a train is in section *P1* and no train is in section *P2*, so the transition *MoveCL* is enabled.

3.2.2 Turnout Component

A semaphore is a concept adopted by computer science from railroad terminology. In our basic Petri Net components we often use *semaphore places* [18]. A semaphore place is a place that controls the routing of tokens in a component, so they are typically used to control the routing of trains. Figure 3.8 on the next page is a Coloured Petri Net model of a *turnout.* It consists of three segment places, *Join, Left* and *Right*, representing respectively the stem, left and right branches of the turnout. It has the same basic structure as the road component, allowing only one train in each segment place. For readability, the arc inscriptions for tokens with type *NoTrain* are omitted. A train can only go from the stem entrance to the left segment place if the turnout has control over its left branch, and similarly for the right branch. If a train enters a turnout from one of the branches, the points must be positioned accordingly. This routing of trains is controlled by the point machine, here modelled as semaphore places L and R with a constant type:

color Switch = with switch;

To be able to route a train from *Join* to *Left*, a *switch* token must reside in place L so that the transition *Ldir+* is enabled. After the transition fires, the train will be in place *Left* and the turnout will still be in position L. The same applies for routing trains to the right. Each turnout component has an initial position, either left or right, indicated by the state of the mutex pair L and R as either L or R carries a *switch* token initially. The turnout in Figure 3.8 on the following page has an initial token in place L, representing initial control in the left branch.

The position of a turnout can only be altered by adding a token in place *Change* which will enable either transition *SetR* or *SetL* depending on

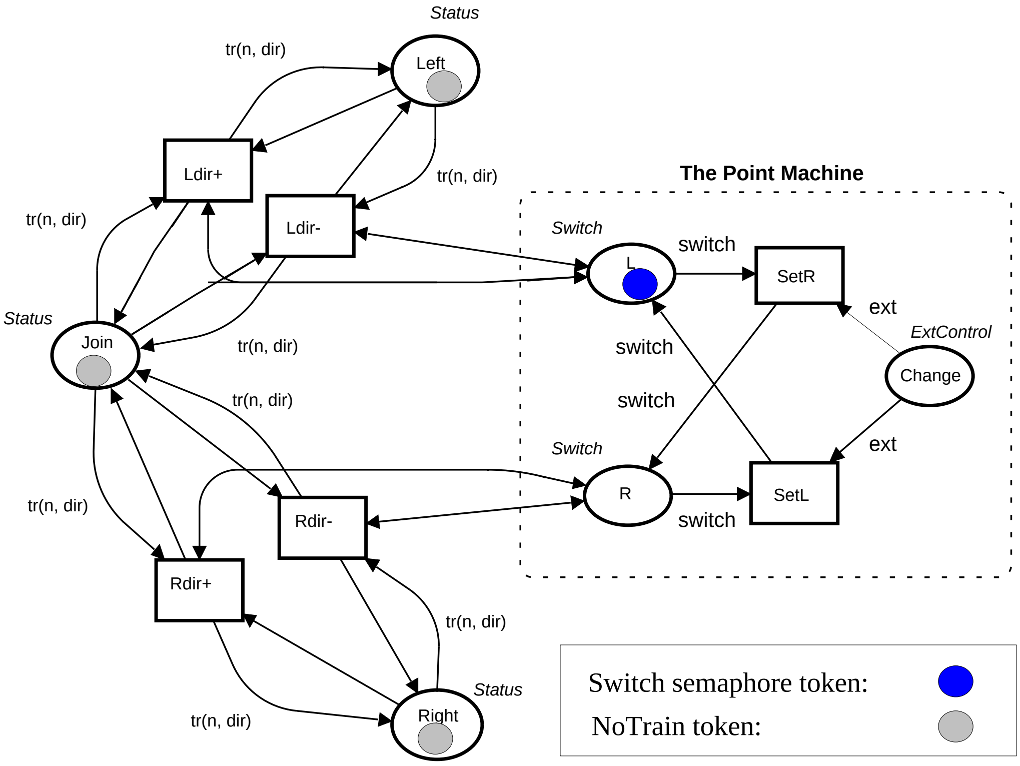


Figure 3.8: A turnout component

where the *switch* token is. Firing of either of these enabled transition will change the points' position. In Figure 3.8, adding a token in place *Change* will enable transition *SetR* and after the transition has fired, a *switch* token is added to *R*, thus changing the controlling branch from left to right. Place *Change* is designed for use with external control, so that the position of the points can be controlled and locked from e.g. the interlocking tower or the control room. *Change* has constant type *ExtControl*:

color ExtControl = with ext;

In subsequent components, all point machines will be constructed as above with the same data structures. The turnout structure is used as the basis of all switch based components. These components use this structure to permit trains to run over one of two tracks, for simplicity we refer to this structure as turnout.

3.2.3 Double Slip

Figure 3.9 is a Petri Net model of the *double slip* in Figure 3.3 on page 21. There are two pairs of points in a double slip and the entrance to a double slip is through one of the side-branches and not from the stem. From each entrance to a double slip there are two exits, controlled by the adjacent point pair.

3.2.4 Rigid Crossing

A *rigid crossing* (Figure 3.10 on page 29) has four segment places *P1, P2, P3* and *P4*, each representing an entrance to the intersection (see Figure 3.4) and four transitions:

- Transition *Move 1* for moving trains from place *P1* to *P3*.
- Transition *Move 2* for moving trains from *P2* to *P4*.
- Transition *Move 3* for moving trains from *P3* to *P1*.
- Transition *Move 4* for moving trains from *P4* to *P2*.

All places carry an initial token of value *NoTrain*.

Since the intersection is a critical region, a semaphore place is introduced to prevent more then one train passing the intersection at the same time, i.e. at most one transition is enabled concurrently. The semaphore place has an initial token which is taken by a train when it enters the crossing and released when it exits.

There are no points in a rigid crossing, and therefore, when a train enters a rigid crossing, it has only one way out of it.

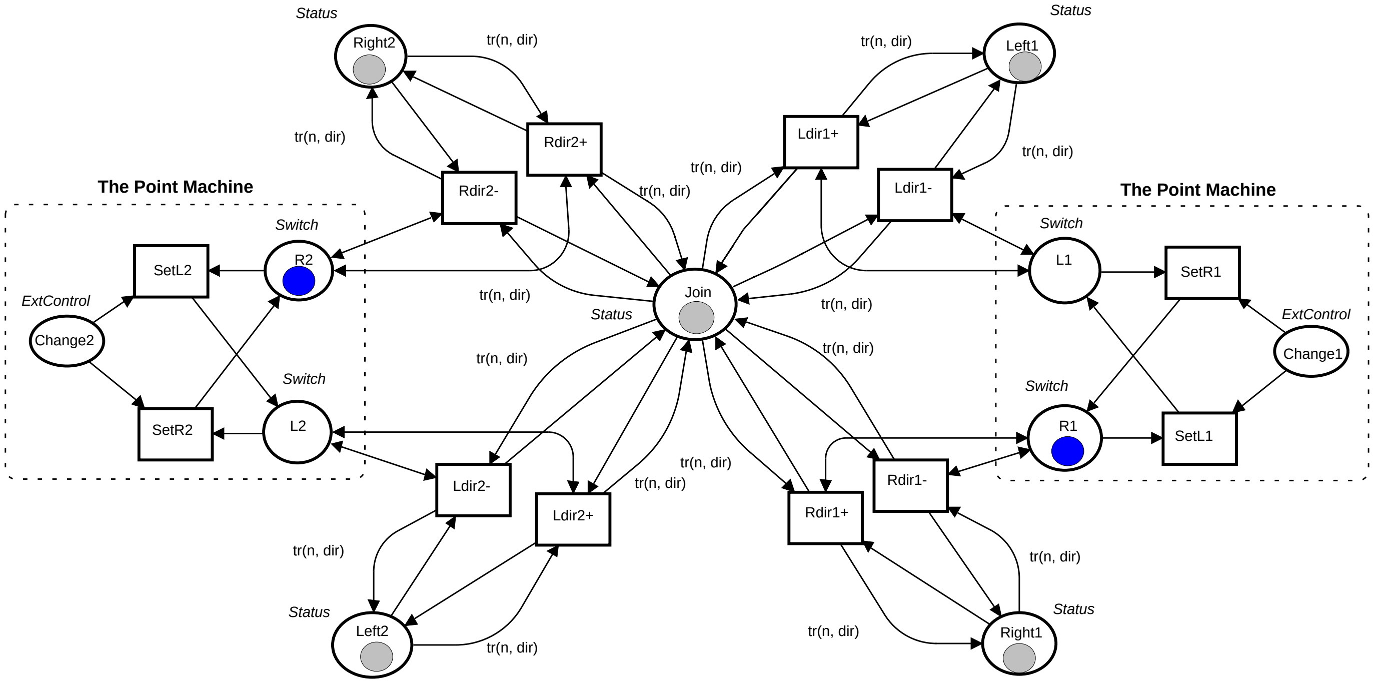


Figure 3.9: A double slip component

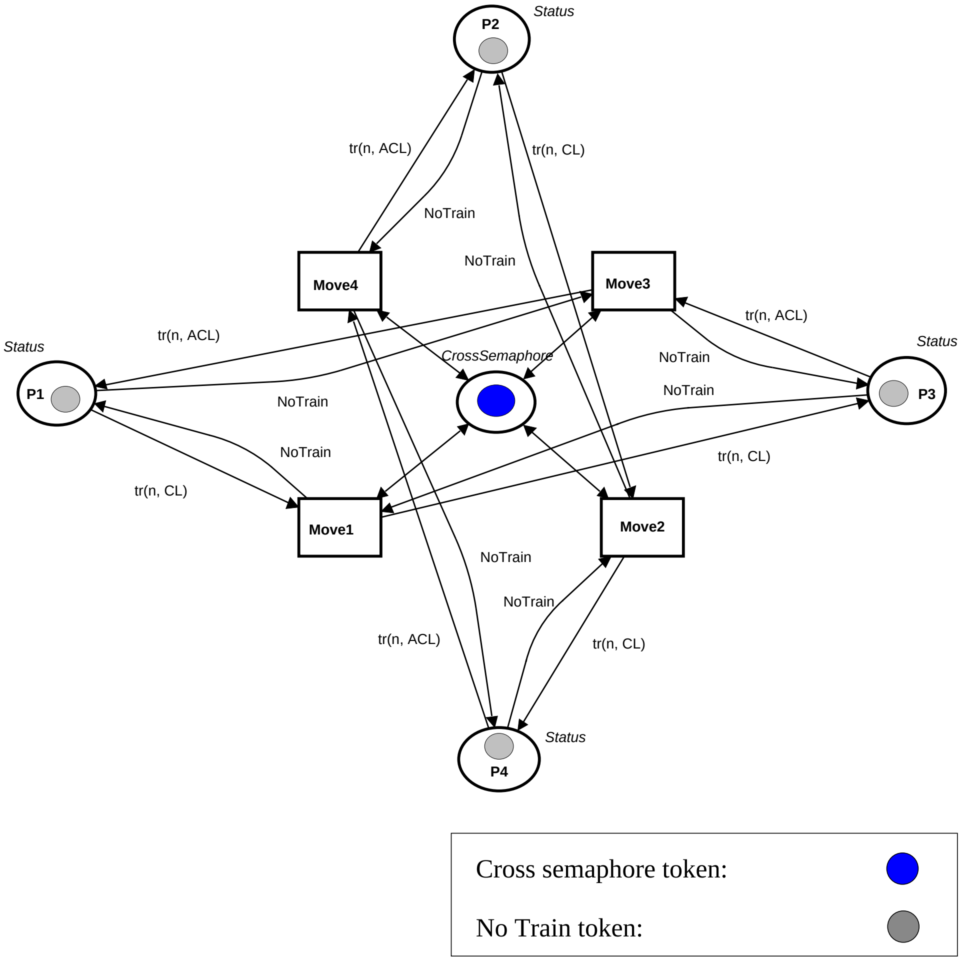


Figure 3.10: A rigid crossing component

3.2.5 Synchronised Scissors

A *scissor*, also called a *double*, is a composition of four turnouts and a crossing, as shown in Figure 3.11.

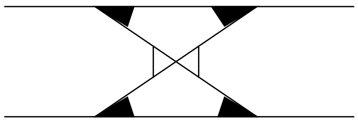


Figure 3.11: A scissor component

Figure 3.12 on the next page shows the scissors modelled as a Petri net component. Each point machine is modelled in the same way as before but the turnouts are integrated with each other in such a way that on the same track line, the right branch of one turnout is the left branch of its adjacent turnout and that for each turnout, its branches are the stems of two of the other turnouts. The center of the component is an integrated rigid crossing modelled as in Figure 3.10 on the preceding page. The initial marking of a scissor has *NoTrain* tokens in each track segment, tokens of type *Switch* in places *L1, R2, L3* and *R4*, indicating the initial position of the points. There is also a semaphore token in the crossing.

Point pairs in a scissor are pair-wise synchronised, so that for two point pairs, changing the position of one also changes the position of the other, i.e. *Change1* is synchronised with *Change3* and *Change2* with *Change4*. Places for synchronisation of point pairs are *ChangeSynchronise1* and *ChangeSynchronise2*. As an example, with the initial marking, a train coming from place *Join1* will be guided to place *Join2*. For the train to move to the adjacent track we need to alter the positions of the points in both turnouts 1 and 3 by adding a token in place *ChangeSynchronise1* which will enable transition *SetSynchronise1*. This changes the positions of both point pairs by adding a token to each pair's *Change*. If it is desired to synchronise all points, a place can be added to control the existing synchroniser (the places *Change-Syncronise*).

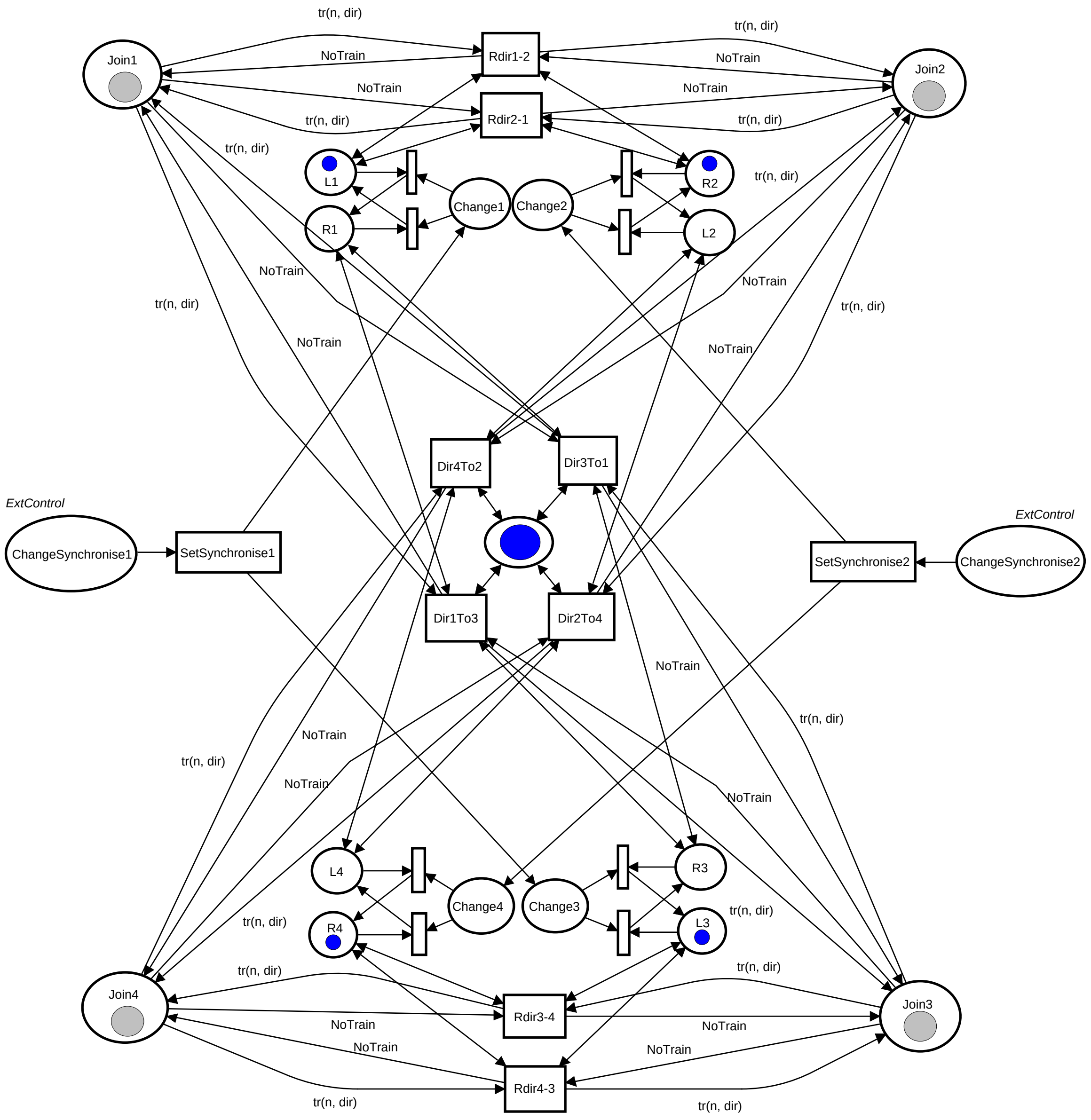


Figure 3.12: Synchronised scissors

3.2.6 Single

A *single* (Figure 3.13 on the next page) is a type of crossover for trains to change to the adjacent track. If it is a crossover to the right, we call it a *single-right* to separate it from singles that cross to the left, called *single-lefts*. Single-rights and -lefts are often combined to form *universals* in railway nets.

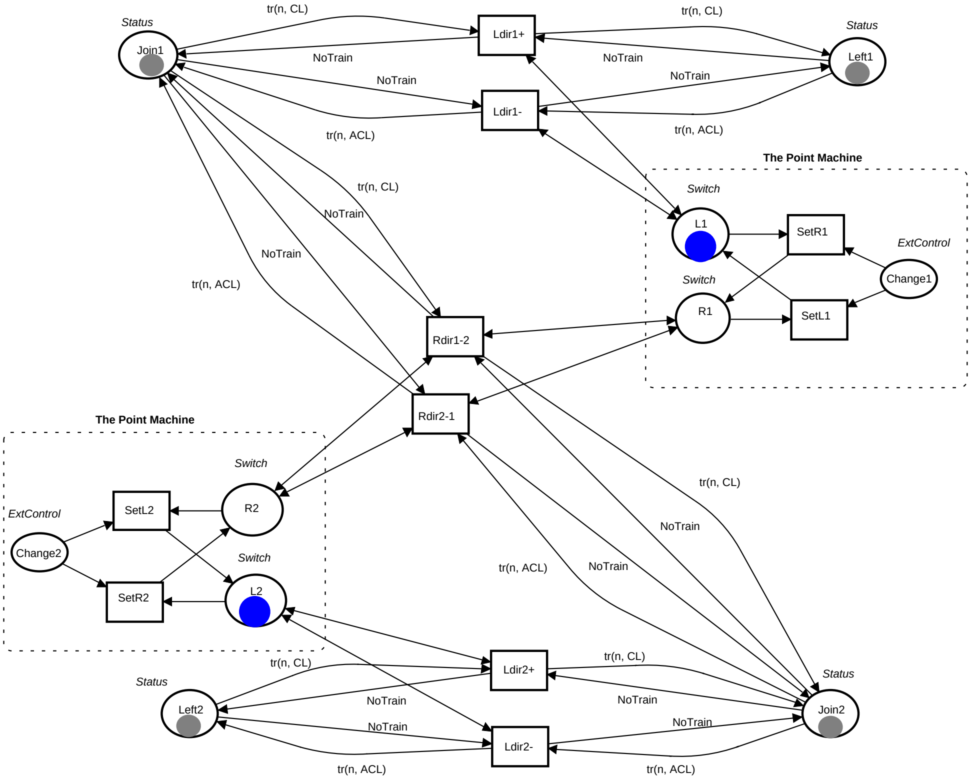


Figure 3.13: A single-right

For points in a single to be synchronised, the approach used in synchronised scissors may be employed.

3.2.7 Routes

A *route* in the domain of railway systems is a description of a way for trains to move across railway network from a start position to a destination. The routes contain information about where the train should drive when encountering turnouts and trains that follow the same train line (normally) have the same route. Taking the routes of trains into account, the data structure of tokens representing the trains can be extended with the type *ListRoute* as follows:

$$\mathit{color\ Train} = \mathit{product\ TrainLineNo} * \mathit{Direction} * \mathit{ListRoute};$$

where

color Branch = with Left | Right | Join;

*color Route = product SwitchNo * Branch;*

color ListRoute = list Route;

The route of a train is given by *ListRoute*, which is a list of pairs of turnout identities and the positions for the points to be in (the way trains are guided through the component). We use the variable $tr(n, dir, r)$ to represent a train with route r.

The turnout in Section 3.2.2 does not consider the routes of trains. When a train enters a turnout, the train will be routed to the branch that is currently in control, even if the train has a route that is not synchronised with the points' position. This means that, for example, if a turnout has control in the left branch but the trains' routes lead to the right, trains will be routed to the left, disregarding their routes. One possible scenario where this can happen, is when a point machine delays to change the points' position. For example, when there are two (or more) trains arriving densely at a turnout and they have different routes in this turnout, then the points may fail to change position in time between these trains, so that two trains with different routes are routed to the same branch.

Now that *ListRoute* constitutes a part of the token structure for trains, correctly routing a train through a turnout therefore depends on the physical points being in the proper position according to the route a train is currently following. Figure 3.14 shows the turnout component in Section 3.2.2, modified for this purpose. We need to identify turnouts uniquely in order to construct a route for each train, so each turnout has a unique identity represented by an Integer token, residing in place *Switch_ID*. Let *sID* be the token holding the identity:

color SwitchNo = int;

var sID : SwitchNo;

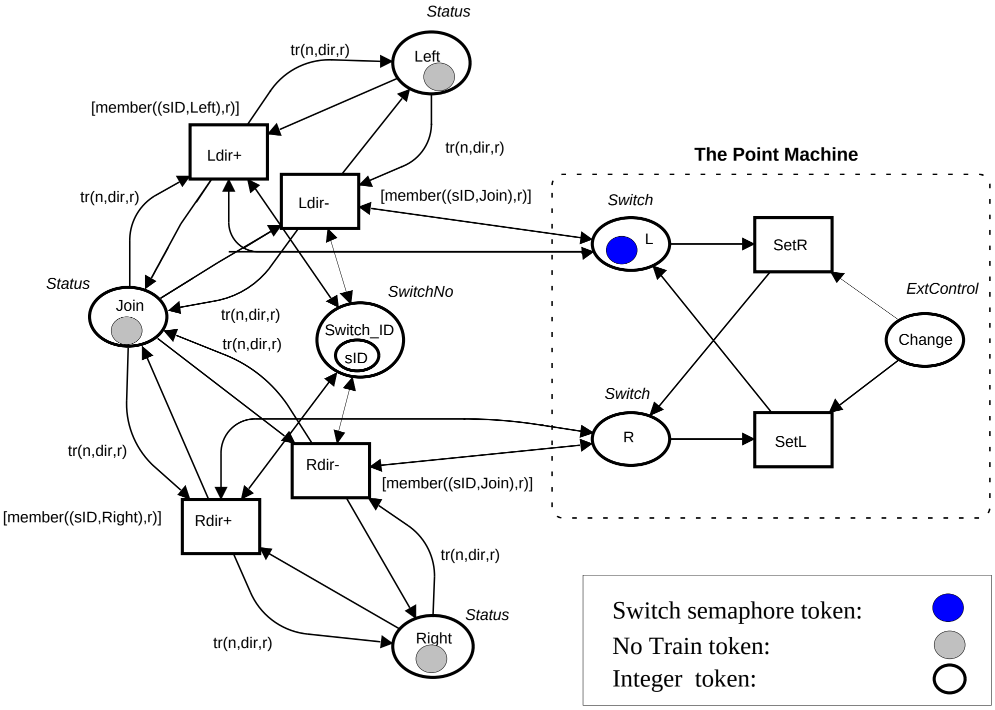


Figure 3.14: A modified turnout

The function *member* searches for routing information from the *ListRoute* attribute, which is a list of tuples containing identities of turnouts and the directions trains are to take in them.

fun member (x, []) = false |
member (x, h::s) = x = h orelse member (x, s);

With a train token in segment place *Join*, the decision whether to go left or right depends on the guards of the transition that routes the train from its current position. If guard $[member((\mathrm{sID}, \mathrm{Left}), r)]$ on transition *Ldir+* evaluates to true, then the train is to be directed to the left branch, or to the right branch if guard $[member((\mathrm{sID}, \mathrm{Right}), r)]$ on transition *Rdir+* evaluates to true. If a train enters from one of the branches, the guard $[member((\mathrm{sID}, \mathrm{Join}), r)]$ must evaluate to true for the train to be able to move to the stem even when there is only one possible way to go. This is because we allow variable *dir* to be evaluated to both *CL* and *ACL*. The guard will prevent a train from entering from the stem to one of the branches and then being routed back, which is possible in the turnout component in Section 3.2.2, giving a possible *livelock*. With these modifications, any train that is to be routed to the branch in which the turnout does not have control, will wait until the points have changed position.

We allow different bindings for variable *dir* in the arc inscriptions $tr(n, dir, r)$, instead of constraining the directions by $tr(n, \mathrm{CL}, r)$ or $tr(n, \mathrm{ACL}, r)$ as in road components. This is because we want to be able to model turnouts on which trains move in opposite directions, with the same component. If we add these constraints, we would have to construct a turnout component for each direction and the component would be topology dependent.

To summarise, a train can only move in a turnout if the points and the travel plan are synchronised (i.e., there is a token in the correct *Switch* place and the corresponding guard evaluates to true) and there is no train in the section ahead. These modifications can be applied to all switch based components, i.e., double slips, scissors and singles, as described here. In the subsequent chapters we will use components with this modifications.

Until now we have shown how track segments, turnouts, double slips, rigid crossings, scissors and singles can be modelled in Petri Nets. We have also shown how trains can be modelled by tokens with data structures. These components will later be used for constructing railway nets.

Chapter 4

Automatic Construction

The process from modelling a railway system in Petri Nets to simulation and analysis is both time consuming and complicated. As the nets tend to be vast and complex and often difficult to handle, we observed the need for abstraction and automatic construction when developing complex Petri Net models, more specifically, models of railroad nets.

We have been looking into a new way of systematically constructing Petri Net models, by introducing an abstraction layer where the system being modelled is specified in a simple language, much closer to the actual systems and require no particular Petri Net knowledge. It is customary to consider the design process as starting with a specification and refining the specification step by step until one reaches an implementation. Our approach differs from this in that we consider the specification as a high level notion, rather then the first step in the process of refinement. With the specification, the corresponding Petri Net implementation is automatically constructed.

In this chapter we will present the foundation for automatic construction of Petri Net models. This allows modelling on an abstract level while generating the Petri Net implementation. Formal theories are specified for both levels and for the actual process of generating Petri Nets. A concrete tool based on this approach will be presented in Chapter 5.

4.1 The Specification Language

The specification language is a graphical language consisting of a set of basic components called *atomic components* that are based on a finite set of nodes $N = \{n_1, n_2, \ldots n_k\}$ and lines $L = N \times N$, a set of interfaces, rules for connecting components and operators that operate on these. We shall now look closer at these different components that constitute the specification language.

4.1.1 Atomic Components

An atomic component $C = \langle L, N^I \rangle$ consists of a set of lines $L \subseteq \langle N \times N \rangle$ and nodes with types, $N^I \subseteq N \times I$. The lines characterise the structure of the component, indicating how nodes are connected. Atomic components are the smallest units in a specification and the set of atomic components is denoted by C^A.

4.1.2 Interfaces and Rules

Nodes in atomic components are either *structural* or *interface* nodes. The structural nodes are internal nodes that are concerned with the internal structure of the component and have no other function. Only interface nodes can participate in a composition with other components. An interface is based on a finite set of distinct *interface types* $I = \{I_1, \ldots, I_n\}$. Each node of a component is equipped with an interface type and components can be connected to each other according to their types and the *composition rules.* These rules vary according to the interface types and prescribe the legal ways to construct composite components. Since structural nodes can not be connected with other components, they have the *empty type* Θ. Rules are defined as follows:

Definition 10 ***Composition rules***
A set of composition rules, written R, is a set of pairs of interface types in I, closed under symmetry such that:

1. $R \subseteq \{\langle I_j, I_k \rangle |\ I_j, I_k \in I\}$
2. $\langle I_j, I_k \rangle \ \in R \Longrightarrow \ \langle I_k, I_j \rangle \ \in R$

It is important to notice that the rules of composition are by default closed under neither reflexivity nor transitivity. In some cases, a reflexive property is useful, but in the railroad case, it may destroy the logic of railroad constructions e.g. the requirement that two end segments can not be connected and that turnouts can not be connected in arbitrary ways. If the composition rules were transitive by default, each component could be connected to any other, hence composition would be too general for the domain of railway system and the composition rules would be vacuous in some cases.

The rules of composition are essential in the process of designing a specification, as the construction must follow certain physical laws and engineering rules that limit the possible combinations. A completely general approach to structural composition is therefore not appropriate for our purpose. The composition rules serve as a guarantee for a syntactically correct specification.

The set of atomic components, interface types and the composition rules constitute the *atomic specification*, $S^A = \langle C^A, I, R\rangle$. Railway specifications are constructed from an initial atomic specification and the compositions of atomic components. These atomic components are the high level representation of basic railroad components.

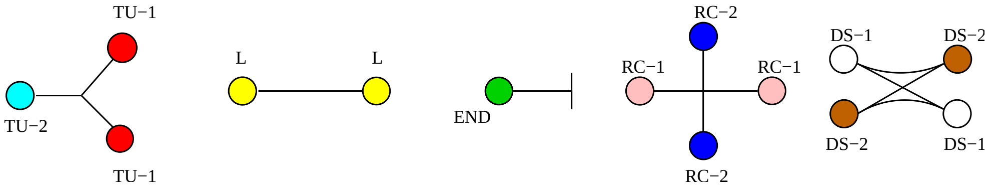
4.1.3 An Example of an Atomic Specification

Figure 4.1 gives an example of a set of atomic components of the railroad components: turnout, line segment, end segment, rigid crossing and double slip. The interface nodes are denoted by colours and types. The table in Figure 4.1 shows the set of distinct interface types, these are the basis for the rules in Figure 4.2 and these rules determine the nodes' legal connections with other components. Here, two turnouts can not be connected to each other by their *TU-1* interfaces and an end segment can only be connected to a line segment.

The rigid crossing component's structure is given by
$C_{RC} = \langle L_{RC}, N^I_{RC}\rangle$, where

$N^I_{RC} = \{\langle n_1, \Theta\rangle, \langle n_2, \text{RC-1}\rangle, \langle n_3, \text{RC-2}\rangle, \langle n_4, \text{RC-1}\rangle, \langle n_5, \text{RC-2}\rangle\}$ and
$L_{RC} = \{\langle n_1, n_2\rangle, \langle n_1, n_3\rangle, \langle n_1, n_4\rangle, \langle n_1, n_5\rangle\}$.

The internal node n_1 with the empty type Θ is not visible in its component because it is a structural node.



Components	*Interface Types*	
Turnout	TU-1	TU-2
Track	L	
End-segment	END	
Rigid crossing	RC-1	RC-2
Double slip	DS-1	DS-2

Figure 4.1: Atomic components and interface types

Reflexive rules		Component rules	
		⟨TU-1,L⟩	⟨TU-2,L⟩
⟨L,L⟩	⟨TU-2,TU-2⟩	⟨L,END⟩	
⟨RC-1,RC-1⟩	⟨RC-2,RC-2⟩	⟨RC-1,L⟩	⟨RC-2,L⟩
⟨DS-1,DS-1⟩	⟨DS-2,DS-2⟩	⟨DS-1, L⟩	⟨DS-2,L⟩

Figure 4.2: Rules for specifications

4.1.4 Composite Specifications

With an initial atomic specification, which is a set of atomic components, a set of interface types and a set of rules, a composite specification can be constructed. The construction of specifications is done through recursive composition, which means building an increasingly complex structure from simple basic components.

A general specification is written $\langle G, S^A\rangle$ where $G = \langle L_G, N_G^I\rangle$ is a connected graph — the structure of the specification — and S^A is an atomic specification so that G is syntactically correct based on the rules in S^A. Two specifications can be joined if there are free interface nodes that match. Informally, an interface node is *free* in a specification if it is not involved in a binding. The result of joining two specifications S_1

and S_2 is a new composite specification, over a concrete binding.

If $i_k \in G$ is a node, then we use I_k to denote the interface type of i_k.

Definition 11 ***Joinable specifications***
Two specifications, $S_1 = \langle G_1, S^A \rangle$ and $S_2 = \langle G_2, S^A \rangle$ over an atomic specification $S^A = \langle C^A, I, R \rangle$, are joinable if there exist free interface nodes $i_1 \in G_1$ and $i_2 \in G_2$ such that $\langle I_1, I_2 \rangle \in R$.

Since the joining between two specifications is through their nodes, we must consider how nodes become a "composite node" and how their types become a "composite type". The names of the nodes in G are distinct, the *replacement function* $\mathrm{sub}(n, m, G)$ replaces a node with name n in G with a new node m. To replace a node in G we must replace the occurrence of this node in both the lines $L \in G$ and the typed nodes $N^I \in G$, which also includes replacing the interface type of this node. Let π_1 and π_2 be the projection functions, $\pi_1(\langle m, n \rangle) = m$ and $\pi_2(\langle m, n \rangle) = n$. The *type-extraction function* ty takes a node and a set of typed nodes and returns the type of this node:

$$\mathrm{ty}(n, N^I) = \pi_2(x) \; if \; x \in N^I \wedge \pi_1(x) = n$$

To access the set of all type assignments, N^I, from a specification, we defined $\mathrm{types}(S) = \pi_2(\pi_1(S))$ that returns all typed nodes in a specification S.

Definition 12 ***Replacement***
The replacement of a node n by a node m in a specification $S = \langle G, S^A \rangle$ is carried out by the replacement function sub.

More specifically, let n, m, x and y denote nodes or interfaces, and let $\langle L_s, N^I_s \rangle$ be a component with lines L_s and typed nodes N^I_s. Let S_1 and S_2 be a specification and N^I be a proper extension of N^I_s. Then

1. $\mathrm{sub}(n, m, n, N^I) = m$
2. $\mathrm{sub}(n, m, x, N^I) = x$ *if* $n \neq x$
3. $\mathrm{sub}(n, m, \langle x, y \rangle, N^I) = \langle \mathrm{sub}(n, m, x, N^I),\ \mathrm{sub}(n, m, y, N^I) \rangle$
4. $\mathrm{sub}(n, \langle i_1, i_2 \rangle, \langle L_s, N^I_s \rangle, N^I) =$
   $\langle \mathrm{sub}(n, \langle i_1, i_2 \rangle, L_s, N^I),$
   $\mathrm{sub}(\langle n, \mathrm{ty}(n, N^I) \rangle, \langle \langle i_1, i_2 \rangle, \langle \mathrm{ty}(i_1, N^I), \mathrm{ty}(i_2, N^I) \rangle \rangle, N^I_s, N^I) \rangle$
   if $n \in \{i_1, i_2\}$

5. $\mathrm{sub}(n, m, S_1 \sqcup S_2, N^I) = \mathrm{sub}(n, m, S_1, N^I) \sqcup \mathrm{sub}(n, m, S_2, N^I)$

6. $\mathrm{sub}(n, m, S, N^I) = \langle \mathrm{sub}(n, m, G, N^I),\ S^A \rangle$

Replacement is defined over the structure of the specification, this is stated in part 6, where S^A remains unchanged, the nodes and types of the atomic components are untouched throughout the recursion. Part 4 performs a replacement in component $\langle L_s, N_s^I \rangle$. The lines are relabelled and nodes are equipped with composite types.

The union of two specifications S_1 and S_2 is the union of their structure and their atomic specifications:

Definition 13 ***Union of specifications***
Let $S_1 = \langle G_1, S_1^A \rangle$ *and* $S_2 = \langle G_2, S_2^A \rangle$ *be two specifications, then the union of* S_1 *and* S_2 *is given by*

$$S_1 \sqcup S_2 = \langle G_1 \cup G_2, S_1^A \cup S_2^A \rangle.$$

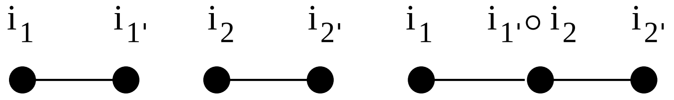
The composition of specifications is denoted with $\sqcap$, and $\sqcap_b$ denotes the concrete binding b. To preserve the information about the composition, the names of the nodes involved in a concrete binding i_1 and i_2 can be combined by concatenation to form a composite name $i_1 \circ i_2$. In the definition of replacement, $i_1 \circ i_2$ is written by the pair $\langle i_1, i_2 \rangle$. With the previous definitions, the composite specification is defined as:

Definition 14 ***Composition***
Let S_1 *and* S_2 *be two specifications, joinable with the binding* $b = [i_1, i_2]$ *over the same atomic specification.*
Let $N^I = \mathrm{types}(S_1) \cup \mathrm{types}(S_2)$ *be the typed nodes in both specifications. Then the composition of* S_1 *and* S_2 *is given by*

$$S_1 \sqcap_b S_2 = \mathrm{sub}(i_1, i_1 \circ i_2, S_1, N^I) \sqcup \mathrm{sub}(i_2, i_1 \circ i_2, S_2, N^I).$$

N^I provides all the interface types and thus ensures that the composite nodes get composite types. Since it does not play any significant role in subsequent proofs, we shall make the reference to N^I implicit by writing $\mathrm{sub}(n, m, S)$. The new node $i_1 \circ i_2$ denotes the joining between S_1 and S_2 into a connected graph. Figure 4.3 on the facing page illustrates the composition of two joinable specifications.

Since composition rules are symmetric, every binding b is equal to the reverse of b. That is, $i_1 \circ i_2 = i_2 \circ i_1$. The node $i_2 \circ i_1$ can be written as



(a) S_1 and S_2 (b) $S_1 \sqcap_{b=[i_{1'}, i_2]} S_2$

Figure 4.3: Composition of specifications

$\overline{i_1 \circ i_2}$ and the reversed binding $\overline{[i_1, i_2]} = [i_2, i_1]$ such that the formula $S_1 \sqcap_b S_2 = S_1 \sqcap_{\overline{b}} S_2$ is fulfilled.

An empty specification is an empty graph, denoted by $\emptyset_s = \langle \emptyset, S^A \rangle$. Composition with the empty specification is always permitted and is through the empty node ϵ, satisfying $n \circ \epsilon = \epsilon \circ n = n$. For empty specifications, $\text{sub}(\epsilon, m, \emptyset_s) = \emptyset_s$ since we need an non-empty specification in order to create nodes.

Lemma 1 *Composition of joinable specifications forms an abelian monoid.*

Proof: Composition $\sqcap$ is an abelian monoid over equality if the following four conditions hold:

1. For all joinable specifications S_1 and S_2, $S_1 \sqcap S_2$ is a specification
2. $S_1 \sqcap_b S_2 = S_2 \sqcap_{\overline{b}} S_1$
3. $(S_1 \sqcap S_2) \sqcap S_3 = S_1 \sqcap (S_2 \sqcap S_3)$
4. $S \sqcap_b \emptyset_s = \emptyset_s \sqcap_{\overline{b}} S = S$

(1) is the closure property and follows from the definition of a correct joinable specification:

If $S_1 = \langle G_1, S^1 \rangle$ and $S_2 = \langle G_2, S^2 \rangle$ are two joinable specifications, then there exists free occurrences of interface nodes $i_1 \in G_1$ and $i_2 \in G_2$ such that $\langle I_1, I_2 \rangle \in R$. The composition of S_1 and S_2 with respect to the concrete binding $b = [i_1, i_2]$ is $\text{sub}(i_1, i_1 \circ i_2, S_1) \sqcup \text{sub}(i_2, i_1 \circ i_2, S_2)$, which is a specification $\langle G_1 \cup G_2, S^A \rangle$.

(2) follows from commutativity of union and that a reversed binding is

the same as the binding itself. Let $b = [i_1, i_2]$:

$$\begin{aligned} S_1 \sqcap_b S_2 &= \mathrm{sub}(i_1, i_1 \circ i_2, S_1) \sqcup \mathrm{sub}(i_2, i_1 \circ i_2, S_2) \\ &\overset{2}{=} \mathrm{sub}(i_2, i_1 \circ i_2, S_2) \sqcup \mathrm{sub}(i_1, i_1 \circ i_2, S_1) \\ &\overset{3}{=} \mathrm{sub}(i_2, i_2 \circ i_1, S_2) \sqcup \mathrm{sub}(i_1, i_2 \circ i_1, S_1) \\ &\overset{4}{=} S_2 \sqcap_{\overline{b}} S_1 \end{aligned}$$

Equation 2 follows from the commutativity of union and equation 3 follows from the equality of reversed names. Equation 4 is the definition of composition.

For (3) We must show $(S_1 \sqcap S_2) \sqcap S_3 = S_1 \sqcap (S_2 \sqcap S_3)$ for distinct binding elements $b_1 = [i_1, i_2]$ and $b_2 = [i_{2'}, i_3]$:

$$(S_1 \sqcap_{[i_1,i_2]} S_2) \sqcap_{[i_{2'},i_3]} S_3$$

$$\overset{1}{=} \mathrm{sub}(i_{2'}, i_{2'} \circ i_3, S_1 \sqcap_{[i_1,i_2]} S_2) \sqcup \mathrm{sub}(i_3, i_{2'} \circ i_3, S_3)$$

$$\overset{2}{=} \mathrm{sub}(i_{2'}, i_{2'} \circ i_3, \mathrm{sub}(i_1, i_1 \circ i_2, S_1)) \sqcup \mathrm{sub}(i_2, i_1 \circ i_2, S_2)) \sqcup \mathrm{sub}(i_3, i_{2'} \circ i_3, S_3)$$

$$\overset{3}{=} \mathrm{sub}(i_1, i_1 \circ i_2, S_1) \sqcup \mathrm{sub}(i_2, i_1 \circ i_2, \mathrm{sub}(i_{2'}, i_{2'} \circ i_3, S_2) \sqcup \mathrm{sub}(i_3, i_{2'} \circ i_3, S_3))$$

$$\overset{4}{=} \mathrm{sub}(i_1, i_1 \circ i_2, S_1) \sqcup \mathrm{sub}(i_2, i_1 \circ i_2, S_2 \sqcap_{[i_{2'},i_3]} S_3)$$

$$\overset{5}{=} S_1 \sqcap_{[i_1,i_2]} (S_2 \sqcap_{[i_{2'},i_3]} S_3)$$

Equations 1, 2, 4 and 5 follow immediately by definition 14. The justification for equation 3 is split in two. First, set union is associative and graph replacement distributes over union. Second, the interface node $i_{2'}$ does not occur free in the specification $\mathrm{sub}(i_1, i_1 \circ i_2, S_1)$ and the interface node i_2 does not occur free in the specification $\mathrm{sub}(i_3, i_{2'} \circ i_3, S_3)$.

In (4) we need to prove the existence of the identity element $\emptyset_s$.
Let $b = [i_1, \epsilon]$:

$$\begin{aligned} S \sqcap_b \emptyset_s &\overset{1}{=} \mathrm{sub}(i_1, i_1 \circ \epsilon, S) \sqcup \mathrm{sub}(\epsilon, i_1 \circ \epsilon, \emptyset_s) \\ &\overset{2}{=} \mathrm{sub}(i_1, i_1, S) \sqcup \mathrm{sub}(\epsilon, i_1, \emptyset_s) \\ &\overset{3}{=} S \sqcup \emptyset_s \\ &\overset{4}{=} S \\ &\overset{5}{=} \emptyset_s \sqcup S \\ &\overset{6}{=} \mathrm{sub}(\epsilon, i_1, \emptyset_s) \sqcup \mathrm{sub}(i_1, i_1, S) \\ &\overset{7}{=} \mathrm{sub}(\epsilon, \epsilon \circ i_1, \emptyset_s) \sqcup \mathrm{sub}(i_1, \epsilon \circ i_1, S) \\ &\overset{8}{=} \emptyset_s \sqcap_{\overline{b}} S \end{aligned}$$

Equations 1 and 8 are again the definition of composition. Equations 2 and 7 follow from the property of the empty node ϵ. Since replacing a node with itself is the identity function, we get $\text{sub}(i_1, i_1, S) = S$, and replacing a node in an empty specification gives the empty specification $\text{sub}(\epsilon, i_1, \emptyset_s) = \emptyset_s$, thus equation 3 and 6 are justified.

■

4.1.5 The Algebra of Decomposition

There are two ways to decompose a specification, split and subtraction. Split, denoted with the symbol $|$, removes a binding in a specification so that the interface nodes involved in the binding become free. The split function does not remove any interface nodes but only frees them as opposed to subtraction, which removes a subset of a specification. Split is based on composition and can only be applied on composite specifications.

Instead of creating a composite node, we want to separate it. The replacement function for decomposition is obtained by replacing part 4 in definition 12 with:

4. $\text{sub}(\langle i_1, i_2\rangle, m, \langle L_s, N_s^I\rangle, N^I) =$
$\langle \text{sub}(\langle i_1, i_2\rangle, m, L_s, N^I),$
$\text{sub}(\langle\langle i_1, i_2\rangle, \langle \text{ty}(\langle i_1, i_2\rangle, N^I)\rangle\rangle, \langle m, \pi_k(\text{ty}(\langle i_1, i_2\rangle, N^I))\rangle, N_s, N^I, \rangle\rangle$
if $m \in \{i_1, i_2\}$ and $k = 1$ if $m = i_1$ else $k = 2$

Definition 15 ***Split***
Let $S_1 \sqcap_b S_2$ be a specification joined with binding $b = [i_1, i_2]$ where $i_1 \in S_1$ and $i_2 \in S_2$ and let $N^I = \text{types}(S_1 \sqcap_b S_2)$ be the typed nodes in $S_1 \sqcap_b S_2$. Then the splitting of $S_1 \sqcap_b S_2$ w.r.t. b is defined as

$$S_1|_b S_2 = \{\text{sub}(i_1 \circ i_2, i_1, S_1, N^I),\ \text{sub}(i_1 \circ i_2, i_2, S_2, N^I)\}.$$

The split function returns a set of specifications. Figure 4.4 on page 47 illustrates splitting of a composite specification.

Lemma 2 *Splitting of a composite specification has the properties:*

1. *$S_1|_b S_2$ is a set of specifications.*
2. $(S_1|_{b_1} S_2)|_{b_2} S_3 = S_1|_{b_1}(S_2|_{b_2} S_3)$

3. $S|_b\emptyset_s = \emptyset_s|_{\overline{b}}S = \{S, \emptyset_s\}$

4. $S_1|_bS_2 = S_2|_{\overline{b}}S_1$

Proof:

(1) follows from the definition of split.

Let $S_1 \sqcap S_2$ be a specification joined with binding $b = [i_1, i_2]$ where $i_1 \in S_1$ and $i_2 \in S_2$. Then the splitting of $S_1 \sqcap_b S_2$ w.r.t b is

$$S_1|_bS_2 = \{\mathrm{sub}(i_1 \circ i_2, i_1, S_1)\ ,\ \mathrm{sub}(i_1 \circ i_2, i_2, S_2)\}$$

The result is a set of specifications $\{S_1, S_2\}$ where
$S_1 = \langle \mathrm{sub}(i_1 \circ i_2, i_1, G_1), S^A\rangle$ and $S_2 = \langle \mathrm{sub}(i_1 \circ i_2, i_2, G_2), S^A\rangle$.

(2) is the associative property:

Let $b_1 = [i_1, i_2]$ and $b_2 = [i_{2'}, i_3]$ where $i_1 \in S_1$, $i_2, i_{2'} \in S_2$ and $i_3 \in S_3$.

$$\begin{aligned}
&(S_1|_{b_1}S_2)|_{b_2}S_3\\
&\overset{1}{=} \{\mathrm{sub}(i_{2'} \circ i_3, i_{2'}, S_1|S_2),\ \mathrm{sub}(i_{2'} \circ i_3, i_3, S_3)\}\\
&\overset{2}{=} \{\mathrm{sub}(i_{2'}\circ i_3, i_{2'},\ \{\mathit{sub}(i_1\circ i_2, i_1, S_1),\ \mathrm{sub}(i_1\circ i_2, i_2, S_2)\}), \mathrm{sub}(i_{2'}\circ i_3, i_3, S_3)\}\\
&\overset{3}{=} \{\mathrm{sub}(i_1\circ i_2, i_1, S_1)\} \cup\{\mathrm{sub}(i_{2'}\circ i_3, i_{2'},\ \mathrm{sub}(i_1\circ i_2, i_2, S_2)),\ \mathrm{sub}(i_{2'}\circ i_3, i_3, S_3)\}\\
&\overset{4}{=} \{\mathrm{sub}(i_1\circ i_2, i_1, S_1),\ \mathrm{sub}(i_1\circ i_2, i_2,\ \mathrm{sub}(i_{2'}\circ i_3, i_{2'}, S_2))\} \cup\{\mathrm{sub}(i_{2'}\circ i_3, i_3, S_3)\}\\
&\overset{5}{=} S_1|(S_2|S_3)
\end{aligned}$$

Equations 1 and 2 are the definition of split. Equation 3 is the result of $i_{2'} \circ i_3$ not being a node in S_1, but in S_2. Equation 4 follows from the fact that $i_1 \circ i_2$ is not a node in S_3 but in S_2.

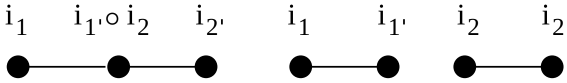
(3) shows the identity element by the immateriality of orders in sets and that a reversed node is the same as the node itself:

$$\begin{aligned}
S|_{b=[i_1,\epsilon]}\emptyset_s &= \{\mathrm{sub}(i_1 \circ \epsilon, i_1, S),\ \mathrm{sub}(i_1 \circ \epsilon, \epsilon, \emptyset_s)\}\\
&= \{\mathrm{sub}(i_1, i_1, S),\ \mathrm{sub}(i_1, \epsilon, \emptyset_s)\}\\
&= \{S, \emptyset_s\}\\
&= \{\emptyset_s, S\} = \{\mathrm{sub}(i_1, \epsilon, \emptyset_s),\ \mathrm{sub}(i_1, i_1, S)\}\\
&= \{\mathrm{sub}(\epsilon \circ i_1, \epsilon, \emptyset_s),\ \mathrm{sub}(\epsilon \circ i_1, i_1, S)\}\\
&= \emptyset_s|_{\overline{b}}S
\end{aligned}$$

(4) is the commutative property:
Let $b = [i_1, i_2]$ where $i_1 \in S_1$ and $i_2 \in S_2$.

$$\begin{aligned} S_1|_b S_2 &= \{\text{sub}(i_1 \circ i_2, i_1, S_1),\ \text{sub}(i_1 \circ i_2, i_2, S_2)\} \\ &= \{\text{sub}(i_1 \circ i_2, i_2, S_2),\ \text{sub}(i_1 \circ i_2, i_1, S_1)\} \\ &= \{\text{sub}(i_2 \circ i_1, i_2, S_2),\ \text{sub}(i_2 \circ i_1, i_1, S_1)\} \\ &= S_2|_{\bar{b}} S_1 \end{aligned}$$

■



(a) $S_1 \sqcap S_2$ (b) $S_1 | S_2$

Figure 4.4: Splitting a specification

The | operator decomposes a composite specification. *Subtraction* is denoted with the symbol ⊟ and removes a specification (which we here refer to as subtrahend) from a composite specification. The ⊟ operator takes a specification as input and returns a specification as output, which means that after a specification has been removed from a composite specification, the remaining specification is one specification and not two or more. This implies that interface nodes participating in compositions with the subtrahend first are to be decomposed (separating them from the subtrahend), and then be composed again, with the subtrahend removed. This second step requires attention on the structure of the subtrahend. If the subtrahend is connected to more than two interface nodes, then after we remove the subtrahend, it is not possible to employ composition on these nodes and achieve one specification. An example of this is shown in Figure 4.5. In 4.5 (a), the subtrahend is connected to three nodes, n_1, n_2 and n_3, and after the subtrahend is removed (4.5 (b)), it is not possible for all these nodes to participate in a composition. This is because composition is a binary operation and that if the number of interface nodes freed in the subtraction was to be an even number larger than two, there would be no way to guarantee that the result would be a single specification. Furthermore, if there was to be more than two interface nodes freed in a subtraction, there would be no way to decide which of these should participate in bindings and in which combinations. For these reasons, a subtrahend is constrained to be isolated by two interface nodes, more specifically, a subtrahend can only connect to the remaining specification by two bindings. Definition 16 formally expresses this condition.

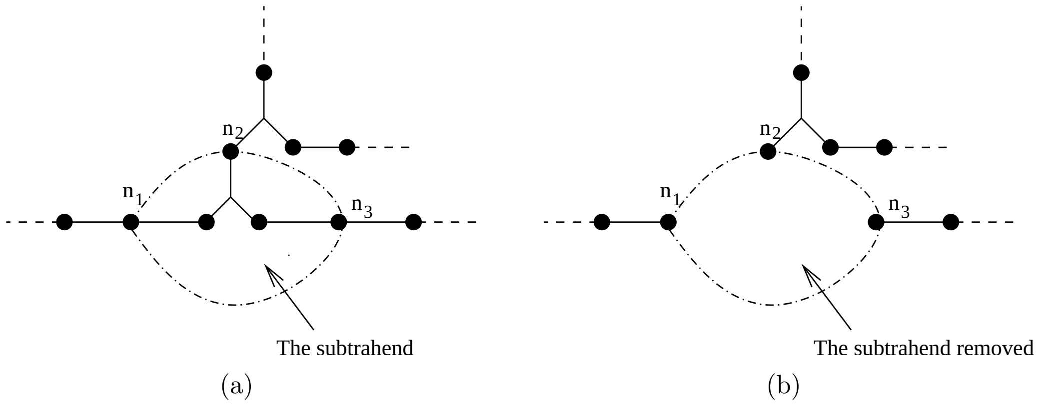


Figure 4.5: A specification that is not isolated

Definition 16 ***Isolation***
Let i be a node, S a specification and $S' = \langle G', S^A \rangle$ are specifications satisfying $\forall L \in G' : L \notin S$. Then S is isolated by two nodes n_1 and n_2 iff

$$\forall i_{\in S} \forall S' (i \in \{n_1, n_2\} \vee i \notin S')$$

Now, we may define subtraction:

Definition 17 ***Subtraction***
Let $S = S_1 \sqcap_{b_1} S_2 \sqcap_{b_2} S_3$ be a specification where $b_1 = [i_1, i_2]$, $b_2 = [i_{2'}, i_3]$ and $N^I = \mathrm{types}(S)$. If $\langle I_1, I_3 \rangle \in R$ and S_2 is isolated by $i_1 \circ i_2$ and $i_{2'} \circ i_3$, then S_2 can be subtracted from S, written $S \boxminus S_2$, as follows:

$$\begin{aligned}&(S_1 \sqcap_{[i_1,i_2]} S_2 \sqcap_{[i_{2'},i_3]} S_3) \boxminus S_2 \\ &\qquad = \mathrm{sub}(i_1 \circ i_2, i_1, S_1, N^I) \ \sqcap_{[i_1,i_3]} \ \mathrm{sub}(i_{2'} \circ i_3, i_3, S_3, N^I).\end{aligned}$$

The precondition requires that the removed specification only connects to the remaining specification by two bindings, specifically, that the removed specification is isolated from the remaining specifications by interface nodes of b_1 and b_2 (and only these) and that the remaining specifications are joinable with interface nodes in b_1 and b_2.

Lemma 3 *Let* S_1, S_2 *and* S_3 *be specifications and let* $S = S_1 \sqcap_{b_1=[i_1,i_2]} S_2 \sqcap_{b_2=[i_{2'},i_3]} S_3$ *where* $\langle I_1, I_3 \rangle \in R$ *and* S_2 *is isolated by* $i_1 \circ i_2$ *and* $i_{2'} \circ i_3$*, then:*

1. $S \boxminus S_2$ *is a specification.*
2. $(S \boxminus S_1) \boxminus S_2 = (S \boxminus S_2) \boxminus S_1$
3. $S \boxminus \emptyset_s = S$
4. $S \boxminus S = \emptyset_s$

Proof:

(1) is the definition of subtraction. By the precondition of subtraction, S_2 is connected to S_1 and S_3 only through two bindings and the interface nodes of S_1 and S_3 that are involved in these bindings are joinable if they become free. Then the result of subtraction by disconnecting these two bindings and then connecting S_1 with S_3 through the same nodes that originally were connected with S_2, gives us a new specification $S_1 \sqcap S_3$ where S_2 is removed.

Given that the precondition of subtraction holds. There are three interesting cases of subtraction of $S_1 \sqcap S_2 \sqcap S_3$. We give the proofs of the closure property of these three cases:

i) If $S_1 = \emptyset_{s_1}$, then:

$$\begin{aligned}
&(S_1 \sqcap_{[\epsilon,i_2]} S_2 \sqcap_{[i_{2'},i_3]} S_3) \boxminus S_2 \\
&\qquad = \mathrm{sub}(\epsilon \circ i_2, \epsilon, \emptyset_{s_1}) \sqcap_{[\epsilon,i_3]} \mathrm{sub}(i_{2'} \circ i_3, i_3, S_3) \\
&\qquad = \mathrm{sub}(\epsilon, \epsilon \circ i_3, \emptyset_{s_1}) \sqcup \mathrm{sub}(i_3, \epsilon \circ i_3, S_3) \\
&\qquad = \emptyset_{s_1} \sqcup S_3 = S_3
\end{aligned}$$

ii) If $S_3 = \emptyset_{s_3}$, then:

$$\begin{aligned}
&(S_1 \sqcap_{[i_1,i_2]} S_2 \sqcap_{[i_{2'},\epsilon]} S_3) \boxminus S_2 \\
&\qquad = \mathrm{sub}(i_1 \circ i_2, i_1, S_1) \sqcap_{[i_1,\epsilon]} \mathrm{sub}(i_{2'} \circ \epsilon, \epsilon, \emptyset_{s_3}) \\
&\qquad = \mathrm{sub}(i_1, i_1 \circ \epsilon, S_1) \sqcup \mathrm{sub}(\epsilon, i_1 \circ \epsilon, \emptyset_{s_3}) \\
&\qquad = S_1 \sqcup \emptyset_{s_3} = S_1
\end{aligned}$$

iii) If S_1, S_2 and S_3 are non-empty, then:

$$\begin{aligned}
&(S_1 \sqcap_{[i_1,i_2]} S_2 \sqcap_{[i_{2'},i_3]} S_3) \boxminus S_2 \\
&\qquad = \mathrm{sub}(i_1 \circ i_2, i_1, S_1) \sqcap_{b=[i_1,i_3]} \mathrm{sub}(i_{2'} \circ i_3, i_3, S_3) \\
&\qquad = \mathrm{sub}(i_1, i_1 \circ i_3, S_1) \sqcup \mathrm{sub}(i_3, i_1 \circ i_3, S_3) \\
&\qquad = S_1 \sqcap_{[i_1,i_3]} S_3
\end{aligned}$$

(2) $(S \boxminus S_1) \boxminus S_2 = (S \boxminus S_2) \boxminus S_1$. Here we assume S_1 does not participate in any other bindings than $b_1 = [i_1, i_2]$. The proof uses the definition of subtraction and composition with empty specifications, so that the subtrahend is always surrounded by two specifications.

$$\begin{aligned}
(S \boxminus S_1) \boxminus S_2 &\overset{1}{=} ((S_1 \sqcap_{[i_1,i_2]} S_2 \sqcap_{[i_{2'},i_3]} S_3) \boxminus S_1) \boxminus S_2 \\
&\overset{2}{=} ((\emptyset_s \sqcap_{[\epsilon,i_{1'}]} S_1 \sqcap_{[i_1,i_2]} (S_2 \sqcap_{[i_{2'},i_3]} S_3)) \boxminus S_1) \boxminus S_2 \\
&\overset{3}{=} (\emptyset_s \sqcap_{[\epsilon,i_2]} \mathrm{sub}(i_1 \circ i_2, i_2, S_2 \sqcap_{[i_{2'},i_3]} S_3)) \boxminus S_2 \\
&\overset{4}{=} (\emptyset_s \sqcap_{b_\epsilon=[\epsilon,i_2]} \mathrm{sub}(i_1 \circ i_2, i_2, S_2) \sqcap_{[i_{2'},i_3]} S_3) \boxminus S_2 \\
&\overset{5}{=} \emptyset_s \sqcap_{[\epsilon,i_3]} \mathrm{sub}(i_{2'} \circ i_3, i_3, S_3) \\
&\overset{6}{=} S_3 \\
&\overset{7}{=} (\emptyset_s \sqcap_{[\epsilon,i_{1'}]} \mathrm{sub}(i_1 \circ i_2, i_1, S_1) \sqcap_{[i_1,i_3]} \mathrm{sub}(i_{2'} \circ i_3, i_3, S_3)) \boxminus S_1 \\
&\overset{8}{=} (\mathrm{sub}(i_1 \circ i_2, i_1, S_1) \sqcap_{[i_1,i_3]} \mathrm{sub}(i_{2'} \circ i_3, i_3, S_3)) \boxminus S_1 \\
&\overset{9}{=} ((S_1 \sqcap_{[i_1,i_2]} S_2 \sqcap_{[i_{2'},i_3]} S_3) \boxminus S_2) \boxminus S_1 \\
&\overset{10}{=} (S \boxminus S_2) \boxminus S_1
\end{aligned}$$

Equations 2 and 8 introduced empty specifications to enclose the subtrahend, S_1, between two specifications. Equations 3 and 5 are the definition of subtraction and that $i_1 \circ i_2$ is not in S_3 justifies equation 4. Equation 6 is proved in Lemma 1 (part 4), which is also used in equation 7 together with the definition of subtraction.

The proof is a special case where S_1 is a "leaf" component. However, it can be generalised by replacing the empty specification, $\emptyset_s$, with e.g. variable S_k, so that if S_1 is a leaf, then S_k is an empty specification. Otherwise, S_k is the specification S_1 is connected to. With this generalisation, the result is $S_k \sqcap S_3$.

For (3):

$$\begin{aligned}
(S_1 \sqcap \emptyset_{s_2} \sqcap \emptyset_{s_3}) \boxminus \emptyset_{s_2} &= \mathrm{sub}(i_1 \circ \epsilon, i_1, S_1) \sqcap \mathrm{sub}(\epsilon, \epsilon, \emptyset_{s_3}) \\
&= S_1
\end{aligned}$$

Note that (3) is not a proof of the existence of an identity element since an empty specifications does not include a non-empty specification. Therefore $\emptyset_s \boxminus S$ is not defined, so $\forall S, S \boxminus \emptyset_s$ do not yield $\emptyset_s \boxminus S$.

(4) shows that there is an inverse or reciprocal of each specification.

$$\begin{aligned}(\emptyset_{s_1} \sqcap S_2 \sqcap \emptyset_{s_3}) \boxminus S_2 &= \mathrm{sub}(\epsilon \circ i_2, \epsilon, \emptyset_{s_1}) \sqcap \mathrm{sub}(i_{2'} \circ \epsilon, \epsilon, \emptyset_{s_3}) \\ &= \emptyset_{s_1} \sqcup \emptyset_{s_3} = \emptyset_s\end{aligned}$$

The inverse element for each specification is the specification itself.

■

Subtraction ($\boxminus$) is definable from split ($|$) and composition ($\sqcap$). This means that the subtraction operator can be replaced by splits and joins of specifications. We use the prefix notation of the composition operator, so $S_1 \sqcap S_2$ can be written $\sqcap(S_1, S_2)$, and the following proof justifies this:

Proof:

Let $S = S_1 \sqcap_{[i_1,i_2]} S_2 \sqcap_{[i_{2'},i_3]} S_3$ be a general specification and $b = [i_1, i_3]$, then:

$$\begin{aligned}\sqcap_b((S_1|S_2|S_3)\backslash S_2) &\overset{1}{=} \sqcap_b(\{\mathrm{sub}(i_1 \circ i_2, i_1, S_1),\ \mathrm{sub}(i_{2'} \circ i_3, i_{2'},\ \mathrm{sub}(i_1 \circ i_2, i_2, S_2)), \\ &\qquad \mathrm{sub}(i_{2'} \circ i_3, i_3, S_3)\}\backslash S_2) \\ &\overset{2}{=} \sqcap_b(\{\mathrm{sub}(i_1 \circ i_2, i_1, S_1),\ \mathrm{sub}(i_{2'} \circ i_3, i_3, S_3)\}) \\ &\overset{3}{=} S_1 \sqcap_{[i_1,i_3]} S_3\end{aligned}$$

The result $S_1 \sqcap_{[i_1,i_3]} S_3$, equals the definition of subtraction:

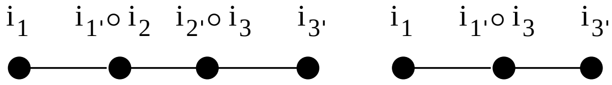
$$S \boxminus S_2 = (S_1 \sqcap_{[i_1,i_2]} S_2 \sqcap_{[i_{2'},i_3]} S_3) \boxminus S_2$$

Hence we have proved that subtraction is definable from split and composition.

Equation 1 uses the definition of split. Equation 2 follows from the set difference, by subtracting S_2, and equation 3 is the infix notation of composition.

■

Figure 4.6 on the following page shows the employment of subtraction on a composite specification.



(a) $S = S_1 \sqcap S_2 \sqcap S_3$ (b) $S \boxminus S_2$

Figure 4.6: Subtraction

4.2 Petri Nets and Algebra

The approach presented here can be applied to many different Petri Net dialects, e.g. Place Transition nets, Coloured Petri Nets, Timed Petri nets etc. To not favour any particular Petri Net dialect, we will consider the most general Petri Net, which all Petri Net dialects are based on. A (general) Petri Net is a triple, $\langle P, T, A\rangle$, where P is a finite set of places, T is a finite set of transitions and A is a finite set of arcs.

Similar to the interface nodes in the specification language, some places in a Petri Net model can participate in a connection with another Petri Net model. We denote these places *interface places.* Interface types and rules are constructed on the specification level and Petri Nets inherit these, therefore, they are not considered here.

4.2.1 The Composition of Petri Nets

As with the composition of specifications, a formal theory for the composition of Petri Net components is specified. It resembles the composition in the specification layer, but applies to Petri Nets. Note that the composition of two Petri Nets in this context is only through interface places. It is not allowed to join two Petri Nets by standard Petri Net semantics such as connecting transitions with places by arcs. The composition of two Petri Nets is thus a composition of two Petri Net models where they connect through interface places. Any Petri Net with interface places can be composed with the empty net $\emptyset_{PN}$, defined as the empty triple.

Definition 18 ***Joinable Petri Nets***
Two Petri Nets $PN_1 = \langle P_1, T_1, A_1\rangle$ *and* $PN_2 = \langle P_2, T_2, A_2\rangle$ *are joinable over a set of interface rules R, if there exists free interface places* $p_1 \in P_1$ *and* $p_2 \in P_2$*, of interface types respectively* I_1 *and* I_2*, such that* $\langle I_1, I_2\rangle \in R$.

When connecting two Petri Net interface places, the input and output arcs that are attached to these places must be redirected to and from the new unique place that arose as a result of merging the two interface places. The redirection is done by the *replacement function* sub, such that the new place replaces the originally non-connected interface places. Observe that transitions are left unchanged, since replacement only changes the place and then redirects the arcs from transitions to the new place.

Definition 19 ***Replacement***
Let p, s, x and y denote places or transitions. The replacement of a place p by a place s in a Petri Net $PN = \langle P, T, A\rangle$ written $\mathrm{sub}(p, s, PN)$, is defined as:

1. $\mathrm{sub}(p, s, p) \ = p$
2. $\mathrm{sub}(p, s, x) \ = x$ *if* $p \neq x$
3. $\mathrm{sub}(p, s, \langle x, y\rangle) = \langle \mathrm{sub}(p, s, x), \mathrm{sub}(p, s, y)\rangle$
4. $\mathrm{sub}(p, s, PN_1 \sqcup PN_2) \ = \mathrm{sub}(p, s, PN_1) \sqcup \mathrm{sub}(p, s, PN_2)$
5. $\mathrm{sub}(p, s, PN) \ = \langle \mathrm{sub}(p, s, P), T, \mathrm{sub}(p, s, A)\rangle$

The composition between Petri Net models is denoted by $\bowtie$, and $\bowtie_b$ denotes composition of Petri Nets with the specific binding b. Union of Petri Nets is the union of their places, transitions and arcs, $PN_1 \sqcup PN_2$ denotes the union of two Petri Nets, PN_1 and PN_2.

Definition 20 ***Composition of Petri Nets***
Let PN_1 and PN_2 be two Petri Nets joinable with the binding $b = [p_1, p_2]$ where $p_1 \in P_1$ and $p_2 \in P_2$. Then the composition of PN_1 and PN_2, is given by

$$PN_1 \bowtie_b PN_2 = \mathrm{sub}(p_1, p_1 \circ p_2, PN_1) \sqcup \mathrm{sub}(p_2, p_1 \circ p_2, PN_2).$$

Lemma 4 *Composition of joinable Petri Nets forms an abelian monoid.*

Proof: We need to prove the following:

1. For all joinable Petri Nets PN_1 and PN_2, $PN_1 \bowtie PN_2$ is a Petri Net
2. $PN_1 \bowtie_b PN_2 = PN_2 \bowtie_{\overline{b}} PN_1$
3. $(PN_1 \bowtie PN_2) \bowtie PN_3 = PN_1 \bowtie (PN_2 \bowtie PN_3)$
4. $PN \bowtie_b \emptyset_{PN} = \emptyset_{PN} \bowtie_{\overline{b}} PN = PN$

(1) Given two Petri Nets $PN_1 = \langle P_1, T_1, A_1 \rangle$ and $PN_2 = \langle P_2, T_2, A_2 \rangle$ that are joinable, then by the definition of joinable Petri Nets there exist interfaces $p_1 \in PN_1$ and $p_2 \in PN_2$ of interface types I_1 and I_2 such that $\langle I_1, I_2 \rangle \in R$. The composition of PN_1 and PN_2 with respect to the binding $b = [p_1, p_2]$ is:

$$\begin{aligned}
\mathrm{sub}(p_1, p_1 \circ p_2, &PN_1) \sqcup \mathrm{sub}(p_2, p_1 \circ p_2, PN_2) \\
&= \langle \mathrm{sub}(p_1, p_1 \circ p_2, P_1), T_1, \mathrm{sub}(p_1, p_1 \circ p_2, A_1) \rangle \\
&\quad \sqcup \ \langle \mathrm{sub}(p_2, p_1 \circ p_2, P_2), T_2, \mathrm{sub}(p_2, p_1 \circ p_2, A_2) \rangle \\
&= \langle \mathrm{sub}(p_1, p_1 \circ p_2, P_1) \cup \mathrm{sub}(p_2, p_1 \circ p_2, P_2), \\
&\quad T_1 \cup T_2, \mathrm{sub}(p_1, p_1 \circ p_2, A_1) \cup \mathrm{sub}(p_2, p_1 \circ p_2, A_2) \rangle
\end{aligned}$$

which is a composite Petri Net.

(2) follows from the commutativity of union and that a reversed binding is the same as the binding itself:

Let $PN_1 = \langle P_1, T_1, A_1 \rangle$ and $PN_2 = \langle P_2, T_2, A_2 \rangle$ be two Petri Nets, and $p_1 \in PN_1$ and $p_2 \in PN_2$ be interface places of types I_1 and I_2 where $\langle I_1, I_2 \rangle \in R$. Then:

$$\begin{aligned}
PN_1 \bowtie_b PN_2 &\overset{1}{=} \mathrm{sub}(p_1, p_1 \circ p_2, PN_1) \sqcup \mathrm{sub}(p_2, p_1 \circ p_2, PN_2) \\
&\overset{2}{=} \mathrm{sub}(p_2, p_1 \circ p_2, PN_2) \sqcup \mathrm{sub}(p_1, p_1 \circ p_2, PN_1) \\
&\overset{3}{=} \mathrm{sub}(p_2, p_2 \circ p_1, PN_2) \sqcup \mathrm{sub}(p_1, p_2 \circ p_1, PN_1) \\
&\overset{4}{=} PN_2 \bowtie_{\overline{b}} PN_1
\end{aligned}$$

Equations 1 and 4 are the definition of composition. Equations 2 and 3 follow from respectively the commutativity of union and the equality of reversed names.

For (3) we must show the associative property:

$(PN_1 \bowtie PN_2) \bowtie PN_3 = PN_1 \bowtie (PN_2 \bowtie PN_3)$ for distinct binding elements $b_1 = [p_1, p_2]$ and $b_2 = [p_{2'}, p_3]$:

$$
\begin{aligned}
&(PN_1 \bowtie_{[p_1,p_2]} PN_2) \bowtie_{[p_{2'},p_3]} PN_3 \\
&\quad \overset{1}{=} \mathrm{sub}(p_{2'}, p_{2'} \circ p_3, PN_1 \bowtie_{[p_1,p_2]} PN_2) \sqcup \mathrm{sub}(p_3, p_{2'} \circ p_3, PN_3) \\
&\quad \overset{2}{=} \mathrm{sub}(p_{2'}, p_{2'} \circ p_3, \mathrm{sub}(p_1, p_1 \circ p_2, PN_1) \\
&\qquad \sqcup \mathrm{sub}(p_2, p_1 \circ p_2, PN_2)) \sqcup \mathrm{sub}(p_3, p_{2'} \circ p_3, PN_3) \\
&\quad \overset{3}{=} \mathrm{sub}(p_1, p_1 \circ p_2, PN_1) \sqcup \mathrm{sub}(p_2, p_1 \circ p_2, \mathrm{sub}(p_{2'}, p_{2'} \circ p_3, PN_2) \\
&\qquad \sqcup \mathrm{sub}(p_3, p_{2'} \circ p_3, PN_3)) \\
&\quad \overset{4}{=} \mathrm{sub}(p_1, p_1 \circ p_2, PN_1) \sqcup \mathrm{sub}(p_2, p_1 \circ p_2, PN_2 \bowtie_{[p_{2'},p_3]} PN_3) \\
&\quad \overset{5}{=} PN_1 \bowtie_{[p_1,p_2]} (PN_2 \bowtie_{[p_{2'},p_3]} PN_3)
\end{aligned}
$$

Equations 1, 2, 4 and 5 follow immediately from definition 20, composition of Petri Nets. Equation 3 follows from that $p_{2'}$ does not occur in $\mathrm{sub}(p_1, p_1 \circ p_2, PN_1)$ and that p_2 does not occur in $\mathrm{sub}(p_3, p_{2'} \circ p_3, PN_3)$. Therefore, PN_2 may connect with PN_3 before connecting with PN_1 and vice versa.

(4) is to prove the identity element for all PN. Let $b = [p_1, \epsilon]$:

$$
\begin{aligned}
PN \bowtie_b \emptyset_{PN} &= \mathrm{sub}(p_1, p_1 \circ \epsilon, PN) \sqcup \mathrm{sub}(\epsilon, p_1 \circ \epsilon, \emptyset_{PN}) \\
&= \mathrm{sub}(p_1, p_1, PN) \sqcup \mathrm{sub}(\epsilon, p_1, \emptyset_{PN}) \\
&= PN \sqcup \emptyset_{PN} \\
&= \langle P \cup \emptyset_p, T \cup \emptyset_T, A \cup \emptyset_A \rangle \\
&= PN \\
&= \langle \emptyset_P \cup P, \emptyset_T \cup T, \emptyset_A \cup A \rangle \\
&= \emptyset_{PN} \sqcup PN \\
&= \mathrm{sub}(\epsilon, p_1, \emptyset_{PN}) \sqcup \mathrm{sub}(p_1, p_1, PN) \\
&= \mathrm{sub}(\epsilon, \epsilon \circ p_1, \emptyset_{PN}) \sqcup \mathrm{sub}(p_1, \epsilon \circ p_1, PN) \\
&= \emptyset_{PN} \bowtie_{\overline{b}} PN
\end{aligned}
$$

■

The results follows from the definitions. Since replacing a place with itself is the identity function we get $\mathrm{sub}(p_1, p_1, PN) = PN$, and replacements in an empty Petri Net gives the empty net, thus $\mathrm{sub}(\epsilon, p_1, \emptyset_{PN}) = \emptyset_{PN}$.

4.2.2 The Decomposition of Petri Nets

Splitting a composite Petri Net is done by removing binding elements so that interface places become free again. The place that is a join between two Petri Nets becomes two free interface places as before the composition, each with a unique name. The arcs involved are also redirected as a consequence of this.

The split operator '||' applies to a Petri Net and gives a set of Petri Nets:

Definition 21 ***Split***
Let $PN_1 \bowtie_b PN_2$ *be a Petri Net joined with binding* $b = [p_1, p_2]$ *where* $p_1 \in P_1$ *and* $p_2 \in P_2$*. Then the splitting of* $PN_1 \bowtie_b PN_2$ *w.r.t.* b *is defined as*

$$PN_1||_b PN_2 = \{\text{sub}(p_1 \circ p_2, p_1, PN_1)\ ,\ \text{sub}(p_1 \circ p_2, p_2, PN_2)\}.$$

Lemma 5 *Splitting of a composite Petri Net has the properties:*

1. $PN_1||_b PN_2$ *is a set of Petri Nets.*
2. $(PN_1||_{b_1} PN_2)||_{b_2} PN_3 = PN_1||_{b_1}(PN_2||_{b_2} PN_3)$
3. $PN||_b \emptyset_{PN} = \emptyset_{PN}||_{\bar{b}} PN = PN$
4. $PN_1||_b PN_2 = PN_2||_{\bar{b}} PN_1$

Proof:

(1) is the definition of split.

Let $PN_1 \bowtie_b PN_2$ be a Petri Net joined with binding $b = [p_1, p_2]$, where $p_1 \in P_1$ and $p_2 \in P_2$. Then the splitting of $PN_1 \bowtie_b PN_2$ w.r.t b is written:

$$PN_1||_b PN_2 = \{\text{sub}(p_1 \circ p_2, p_1, PN_1),\ \text{sub}(p_1 \circ p_2, p_2, PN_2)\}.$$

The result is a set of Petri Nets $\{PN_1, PN_2\}$ where
$PN_1 = \langle \text{sub}(p_1 \circ p_2, p_1, P_1)), T_1, \text{sub}(p_1 \circ p_2, p_1, A_1)\rangle$ and
$PN_2 = \langle \text{sub}(p_1 \circ p_2, p_2, P_2)), T_2, \text{sub}(p_1 \circ p_2, p_2, A_2)\rangle$.

(2) is the associative property:

Let $b_1 = [p_1, p_2]$ and $b_2 = [p_{2'}, p_3]$ where $p_1 \in PN_1,\ \ p_2, p_{2'} \in PN_2$ and $p_3 \in PN_3$.

$$
\begin{aligned}
&(PN_1||_{b_1}PN_2)||_{b_2}PN_3\\
&\qquad \overset{1}{=} \{\mathrm{sub}(p_{2'} \circ p_3, p_{2'}, PN_1||PN_2), \mathrm{sub}(p_{2'} \circ p_3, p_3, PN_3)\}\\
&\qquad \overset{2}{=} \{\mathrm{sub}(p_{2'} \circ p_3, p_{2'}, \{\mathrm{sub}(p_1 \circ p_2, p_1, PN_1),\\
&\qquad\quad \mathrm{sub}(p_1 \circ p_2, p_2, PN_2)\}), \mathrm{sub}(p_{2'} \circ p_3, p_3, PN_3)\}\\
&\qquad \overset{3}{=} \{\mathrm{sub}(p_1 \circ p_2, p_1, PN_1)\} \cup\\
&\qquad\quad \{\mathrm{sub}(p_{2'} \circ p_3, p_{2'}, \mathrm{sub}(p_1 \circ p_2, p_2, PN_2)), \mathrm{sub}(p_{2'} \circ p_3, p_3, PN_3)\}\\
&\qquad \overset{4}{=} \{\mathrm{sub}(p_1 \circ p_2, p_1, PN_1), \mathrm{sub}(p_1 \circ p_2, p_2, \mathrm{sub}(p_{2'} \circ p_3, p_{2'}, PN_2))\}\\
&\qquad\quad \cup \{\mathrm{sub}(p_{2'} \circ p_3, p_3, PN_3)\}\\
&\qquad \overset{5}{=} PN_1||_{b_1}(PN_2||_{b_2}PN_3)
\end{aligned}
$$

Equations 1, 2 and 5 are the definition of splitting a composite Petri net. Equation 3 is the result of that $p_{2'} \circ p_3$ is not a place in PN_1, but in PN_2. That $p_1 \circ p_2$ is not a place in PN_3 but it is in PN_2 justifies equation 4.

(3) shows the identity element $\emptyset_{PN}$:

$$
\begin{aligned}
PN||_{[p_1,\epsilon]}\emptyset_{PN} &= \{\mathrm{sub}(p_1 \circ \epsilon, p_1, PN), \mathrm{sub}(p_1 \circ \epsilon, \epsilon, \emptyset_{PN})\}\\
&= \{\mathrm{sub}(p_1, p_1, PN), \mathrm{sub}(p_1, \epsilon, \emptyset_{PN}))\}\\
&= \{PN, \emptyset_{PN}\}\\
&= \{\emptyset_{PN}, PN\} = \{\mathrm{sub}(p_1, \epsilon, \emptyset_{PN}), \mathrm{sub}(p_1, p_1, PN)\}\\
&= \{\mathrm{sub}(\epsilon \circ p_1, \epsilon, \emptyset_{PN}), \mathrm{sub}(\epsilon \circ p_1, p_1, PN)\}\\
&= \emptyset_{PN}||_{[\epsilon,p_1]}PN
\end{aligned}
$$

(4) is the commutative property:

Let $b = [p_1, p_2]$ where $p_1 \in PN_1$ and $p_2 \in PN_2$

$$
\begin{aligned}
PN_1||_bPN_2 &= \{\mathrm{sub}(p_1 \circ p_2, p_1, PN_1), \mathrm{sub}(p_1 \circ p_2, p_2, PN_2)\}\\
&= \{\mathrm{sub}(p_1 \circ p_2, p_2, PN_2), \mathrm{sub}(p_1 \circ p_2, p_1, PN_1)\}\\
&= \{\mathrm{sub}(p_2 \circ p_1, p_2, PN_2), \mathrm{sub}(p_2 \circ p_1, p_1, PN_1)\}\\
&= PN_2||_{\overline{b}}PN_1
\end{aligned}
$$

■

Subtraction is also defined for Petri Net components. The operator is denoted $\ominus$, operates on a composite Petri Net and returns a Petri Net. The preconditions for subtraction of Petri Nets are the same as subtraction of specifications, only applied to Petri Nets.

Definition 22 ***Subtraction***
Let $PN = PN_1 \bowtie_{b_1} PN_2 \bowtie_{b_2} PN_3$ *be a Petri Net where* $b_1 = [p_1, p_2]$ *and* $b_2 = [p_{2'}, p_3]$*. If* $\langle I_1, I_3 \rangle \in R$ *and* PN_2 *is isolated by* $p_1 \circ p_2$ *and* $p_{2'} \circ p_3$*, then* PN_2 *can be subtracted from* PN*, written* $PN \ominus PN_2$ *in the following way:*

$$\begin{aligned}(PN_1 \bowtie_{b_1=[p_1,p_2]} PN_2 \bowtie_{b_2=[p_{2'},p_3]} PN_3) \ominus PN_2 \\ = \mathrm{sub}(p_1 \circ p_2, p_1, PN_1) \bowtie_{[p_1,p_3]} \mathrm{sub}(p_{2'} \circ p_3, p_3, PN_3)\end{aligned}$$

Lemma 6 *Let* PN_1, PN_2 *and* PN_3 *be Petri Nets and let*
$PN = PN_1 \sqcap_{b_1=[p_1,p_2]} PN_2 \sqcap_{b_2=[p_{2'},p_3]} PN_3$ *where*
$\langle I_1, I_3 \rangle \in R$ *and* PN_2 *is isolated by* $p_1 \circ p_2$ *and* $p_{2'} \circ p_3$*, then:*

1. $PN \ominus PN_2$ *is a Petri Net.*

2. $(PN \ominus PN_1) \ominus PN_2 = (PN \ominus PN_2) \ominus PN_1$

3. $PN \ominus \emptyset_{PN} = PN$

4. $PN \ominus PN = \emptyset_{PN}$

Proof:

(1) By the precondition of subtraction, PN_2 is connected to PN_1 and PN_3 only through two bindings and the interface places of PN_1 and PN_3 that are involved in these bindings are joinable if they become free. Then, by disconnecting these bindings, we may remove PN_2 and join PN_1 with PN_3. The result is proved to be a composite Petri Net, $PN_1 \bowtie PN_2$, in Lemma 4.

Given that the precondition of subtraction holds, there are three interesting cases of subtraction of $PN_1 \bowtie PN_2 \bowtie PN_3$. We give the closure property of these three cases:

i) If $PN_1 = \emptyset_{PN_1}$, then:

$$\begin{aligned}
&(PN_1 \bowtie_{[\epsilon,p_2]} PN_2 \bowtie_{b_2=[p_{2'},p_3]} PN_3) \ominus PN_2 \\
&\quad = \mathrm{sub}(p_\epsilon \circ p_2, \epsilon, \emptyset_{PN_1}) \bowtie_{[\epsilon,p_3]} \mathrm{sub}(p_{2'} \circ p_3, p_3, PN_3) \\
&\quad = \mathrm{sub}(\epsilon, \epsilon \circ p_3, \emptyset_{PN_1}) \sqcup \mathrm{sub}(p_3, \epsilon \circ p_3, PN_3) \\
&\quad = \emptyset_{PN_1} \sqcup PN_3 \\
&\quad = PN_3
\end{aligned}$$

ii) If $PN_3 = \emptyset_{PN_3}$, then:

$$\begin{aligned}
&(PN_1 \bowtie_{[p_1,p_2]} PN_2 \bowtie_{[p_{2'},\epsilon]} PN_3) \ominus PN_2 \\
&\quad = \mathrm{sub}(p_1 \circ p_2, p_1, PN_1) \bowtie_{[p_1,\epsilon]} \mathrm{sub}(p_{2'} \circ \epsilon, \epsilon, \emptyset_{PN_3}) \\
&\quad = \mathrm{sub}(p_1, p_1 \circ \epsilon, PN_1) \sqcup \mathrm{sub}(\epsilon, p_1 \circ \epsilon, \emptyset_{PN_3}) \\
&\quad = PN_1 \sqcup \emptyset_{PN_3} \\
&\quad = PN_1
\end{aligned}$$

iii) If PN_1, PN_2 and PN_3 are non-empty, then:

$$\begin{aligned}
&(PN_1 \bowtie_{[p_1,p_2]} PN_2 \bowtie_{[p_{2'},p_3]} PN_3) \ominus PN_2 \\
&\quad = \mathrm{sub}(p_1 \circ p_2, p_1, PN_1) \bowtie_{[p_1,p_3]} \mathrm{sub}(p_{2'} \circ p_3, p_3, PN_3) \\
&\quad = \mathrm{sub}(p_1, p_1 \circ p_3, PN_1) \sqcup \mathrm{sub}(p_3, p_1 \circ p_3, PN_3) \\
&\quad = PN_1 \bowtie PN_3
\end{aligned}$$

(2) is to prove $(PN \ominus PN_1) \ominus PN_2 = (PN \ominus PN_2) \ominus PN_1$.

The proof is the same as for subtraction of specifications, with the special case that PN_1 only connects to PN_2 (a leaf Petri Net component), and can also be generalised:

$$\begin{aligned}
&(PN \ominus PN_1) \ominus PN_2 \\
&\quad = ((PN_1 \bowtie_{[p_1,p_2]} PN_2 \bowtie_{[p_{2'},p_3]} PN_3) \ominus PN_1) \ominus PN_2 \\
&\quad = (\emptyset_{PN} \bowtie_{[\epsilon,p_{1'}]} PN_1 \bowtie_{[p_1,p_2]} (PN_2 \bowtie_{[p_{2'},p_3]} PN_3)) \ominus PN_1 \ominus PN_2 \\
&\quad = (\emptyset_{PN} \bowtie_{[\epsilon,p_2]} \mathrm{sub}(p_1 \circ p_2, p_2, PN_2 \bowtie_{[p_{2'},p_3]} PN_3)) \ominus PN_2 \\
&\quad = (\emptyset_{PN} \bowtie_{[\epsilon,p_2]} \mathrm{sub}(p_1 \circ p_2, p_1, PN_2) \bowtie_{[p_{2'},p_3]} PN_3) \ominus PN_2 \\
&\quad = \emptyset_{PN} \bowtie_{[\epsilon,p_3]} \mathrm{sub}(p_{2'} \circ p_3, p_3, PN_3) \\
&\quad = PN_3 \\
&\quad = (\emptyset_{PN} \bowtie_{[\epsilon,p_{1'}]} \mathrm{sub}(p_1 \circ p_2, p_1, PN_1) \bowtie_{[p_1,p_3]} \mathrm{sub}(p_{2'} \circ p_3, p_3, PN_3)) \ominus PN_1 \\
&\quad = (\mathrm{sub}(p_1 \circ p_2, p_1, PN_1) \bowtie_{[p_1,p_3]} \mathrm{sub}(p_{2'} \circ p_3, p_3, PN_3)) \ominus PN_1 \\
&\quad = ((PN_1 \bowtie_{[p_1,p_2]} PN_2 \bowtie_{[p_{2'},p_3]} PN_3) \ominus PN_2) \ominus PN_1 \\
&\quad = (PN \ominus PN_2) \ominus PN_1
\end{aligned}$$

As mentioned earlier for subtraction of specifications, (3) is not the identity property since $\emptyset_{PN} \ominus PN$ is not defined. The result follows from the proof of Lemma 4, part 4.

$$\begin{aligned}(PN_1 \bowtie \emptyset_{PN_2} \bowtie \emptyset_{PN_3}) \ominus \emptyset_{PN_2} &= \text{sub}(p_1 \circ \epsilon, p_1, PN_1) \bowtie \text{sub}(\epsilon, \epsilon, \emptyset_{PN_3}) \\ &= PN_1 \sqcup \emptyset_{PN_3} \\ &= PN_1\end{aligned}$$

(4) shows that there is an inverse of each Petri Net.

$$\begin{aligned}(\emptyset_{PN_1} \bowtie PN_2 \bowtie \emptyset_{PN_3}) \ominus PN_2 &= \text{sub}(\epsilon \circ p_2, \epsilon, \emptyset_{PN_1}) \bowtie \text{sub}(p_{2'} \circ \epsilon, \epsilon, \emptyset_{PN_3}) \\ &= \emptyset_{PN_1} \sqcup \emptyset_{PN_3} \\ &= \emptyset_{PN}\end{aligned}$$

For each Petri Net, the inverse element is the net itself.
■

4.3 Saturation

Saturation is an automatic construction of a Petri Net implementation from a specification. In the process of saturation, a specification, composed of atomic components with respect to composition rules, is taken. This specification, forming a graphical structure as described in Section 4.1, is saturated with a Petri Net implementation.

4.3.1 Atomic Saturation

Saturation associates a high level specification with concrete Petri Nets and can be decomposed into successive compositions of *atomic saturations* — assignments of atomic components to a set of Petri Nets.

Definition 23 ***Atomic saturation***
Let $S^A = \langle C^A, I, R\rangle$ be an atomic specification. An atomic saturation, written $\mathcal{A}$, is a function from a set of atomic components to a set of Petri Nets PN, such that:

$$\forall C \in C^A \exists! P \in PN : (\mathcal{A}(C) = P)$$

An atomic saturation is done by first constructing a library of atomic and Petri Net components and then making an explicit assignment between them. Specifically, it is a bijection from interface nodes to interface places. With an atomic saturation, the implementation can automatically be generated. Figure 4.7 shows an example of an atomic saturation which assigns each atomic high level specification component to a concrete Petri Net.

After an atomic saturation, each component in the specification will be bound to a corresponding Petri Net component and compositions in the specification level will lead to compositions in the Petri Net level.

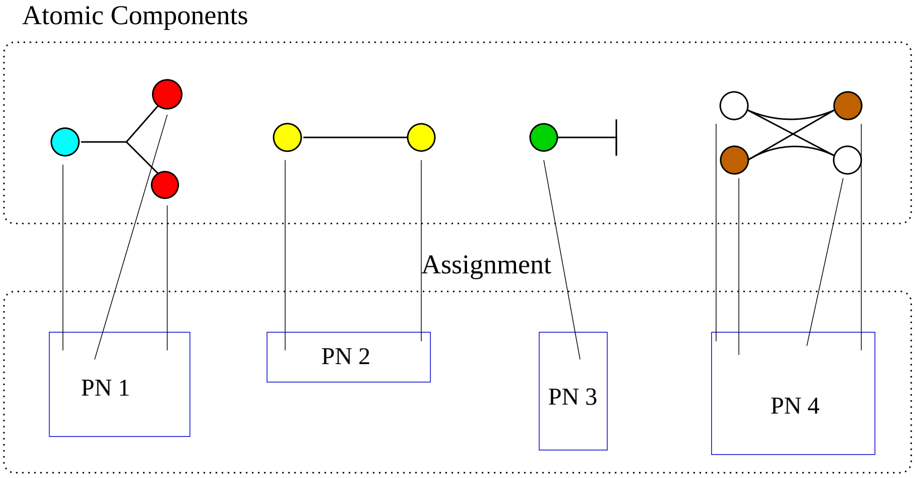


Figure 4.7: Atomic saturation

4.3.2 Theorem of Construction

A composite specification S can be measured by its number of interface bindings, denoted $\mathtt{m}(\mathrm{S})$. We consider components built up sequentially from 1-step interface bindings.

Definition 24 ***Size of specifications***
We define the size of a specification $\mathtt{m}(\mathrm{S})$ *by recursion:*

1. *If* S *is an atomic component, then* $\mathtt{m}(\mathrm{S}) = 0$
2. *if* S *is a composite specification* $S = S_1 \sqcap S_2$*, then* $\mathtt{m}(S_1 \sqcap S_2) = \mathtt{m}(S_1) + \mathtt{m}(S_2) + 1$

We use $\mathtt{i}(S)$ and $\mathtt{i}(N)$ to denote the functions that return the free interface nodes of S and interface places of N. Given a concrete binding b at the specification level, we use $\mathcal{A}(b)$ to refer to the corresponding binding at the Petri Net level, after the bijection of the binding elements.

Definition 25 ***Saturation***
Let $S^{\mathcal{A}}$ be an atomic component, $\mathcal{A}$ an atomic saturation, and S an arbitrary correct specification. Suppose that $S^{\mathcal{A}}$ and S are joinable over an interface binding b. Then we define a saturation function sat *such that:*

(i) $\mathrm{sat}(S^{\mathcal{A}}, \mathcal{A}) \; = \mathcal{A}(S^{\mathcal{A}})$

(ii) $\mathrm{sat}(S^{\mathcal{A}} \sqcap_b S, \mathcal{A}) \; = \mathrm{sat}(S^{\mathcal{A}}, \mathcal{A}) \bowtie_{\mathcal{A}(b)} \mathrm{sat}(S, \mathcal{A})$

Lemma 7 *Let $S_1^{\mathcal{A}}, S_2^{\mathcal{A}}, \ldots, S_{n+1}^{\mathcal{A}}$ be atomic components. Then*

$$\begin{aligned}&\mathrm{sat}(S_1^{\mathcal{A}} \sqcap_{b_1} S_2^{\mathcal{A}} \sqcap_{b_2} \cdots \sqcap_{b_n} S_{n+1}^{\mathcal{A}}, \mathcal{A}) = \\ &\mathrm{sat}(S_1^{\mathcal{A}}, \mathcal{A}) \bowtie_{\mathcal{A}(b_1)} \mathrm{sat}(S_2^{\mathcal{A}}, \mathcal{A}) \bowtie_{\mathcal{A}(b_2)} \cdots \bowtie_{\mathcal{A}(b_n)} \mathrm{sat}(S_{n+1}^{\mathcal{A}}, \mathcal{A})\end{aligned}$$

The lemma captures the actual process of recursively constructing a Petri Net from its specification using saturation.

When a specification S is constructed, it can be saturated with Petri Nets and a concrete implementation of the whole specification can be computed. The theorem of construction states that this output implementation is the same as the result of first saturating parts of the specification and then joining these at the Petri Net level. Formally this means:

Theorem 1 ***The theorem of construction***
Let S_1 and S_2 be two correct specifications that are joinable over an interface binding b, w.r.t. an atomic saturation $\mathcal{A}$. Then

$$\mathrm{sat}(S_1 \sqcap_b S_2), \mathcal{A}) = \mathrm{sat}(S_1, \mathcal{A}) \bowtie_{\mathcal{A}(b)} \mathrm{sat}(S_2, \mathcal{A})$$

Proof: By induction over the size of both S_1 and S_2

Induction basis: $\mathtt{n}(S_1) + \mathtt{n}(S_2) = 0$

This means that both S_1 and S_2 are atomic components, still not connected to others. By the assumption of the theorem, S_1 and S_2 are joinable by the binding b. By the definition of saturation, the specification and the Petri Net implementation have equal interfaces modulo the bijection, so that $\mathbb{i}(S_1^{\mathcal{A}})$ corresponds to $\mathbb{i}(\text{sat}(S_1^{\mathcal{A}}, \mathcal{A}))$ and $\mathbb{i}(S_2^{\mathcal{A}})$ to $\mathbb{i}(\text{sat}(S_2^{\mathcal{A}}, \mathcal{A}))$.

$\text{sat}(S_1^{\mathcal{A}}, \mathcal{A}))$ and $\text{sat}(S_2^{\mathcal{A}}, \mathcal{A})$ are then joinable with the interface binding $\mathcal{A}(b)$, where elements in $\mathcal{A}(b)$ are interface places. Hence

$$\text{sat}(S_1^{\mathcal{A}}, \mathcal{A}) \bowtie_{\mathcal{A}(b)} \text{sat}(S_2^{\mathcal{A}}, \mathcal{A}) = \text{sat}(S_1^{\mathcal{A}} \sqcap_b S_2^{\mathcal{A}}, \mathcal{A})$$

Induction step: $\mathbb{n}(S_1)+\mathbb{n}(S_2) = \text{k} + 1$.

There are two possible cases, either:

(i) $S_1 = S_1^{'} \sqcap_e S_1^{\mathcal{A}}$ where $S_1^{\mathcal{A}}$ is an atomic component, or

(ii) $S_2 = S_2^{'} \sqcap_e S_2^{\mathcal{A}}$ where $S_2^{\mathcal{A}}$ is an atomic component.

Consider (i):

$$\begin{aligned}
\text{sat}(S_1 \sqcap_b S_2, \mathcal{A}) &= \text{sat}((S_1^{'} \sqcap_e S_1^{\mathcal{A}}) \sqcap_b S_2, \mathcal{A}) \\
&\overset{2}{=} \text{sat}(S_2 \sqcap_{\overline{b}} (S_1^{'} \sqcap_e S_1^{\mathcal{A}}), \mathcal{A}) \\
&\overset{3}{=} \text{sat}((S_2 \sqcap_{\overline{b}} S_1^{'}) \sqcap_e S_1^{\mathcal{A}}, \mathcal{A}) \\
&\overset{4}{=} \text{sat}(S_2 \sqcap_{\overline{b}} S_1^{'}, \mathcal{A}) \bowtie_{\mathcal{A}(e)} \text{sat}(S_1^{\mathcal{A}}, \mathcal{A}) \\
&\overset{5}{=} (\text{sat}(S_2, \mathcal{A}) \bowtie_{\mathcal{A}(\overline{b})} \text{sat}(S_1^{'}, \mathcal{A})) \bowtie_{\mathcal{A}(e)} \text{sat}(S_1^{\mathcal{A}}, \mathcal{A}) \\
&\overset{6}{=} \text{sat}(S_2, \mathcal{A}) \bowtie_{\mathcal{A}(\overline{b})} (\text{sat}(S_1^{'}, \mathcal{A}) \bowtie_{\mathcal{A}(e)} \text{sat}(S_1^{\mathcal{A}}, \mathcal{A})) \\
&\overset{7}{=} \text{sat}(S_2, \mathcal{A}) \bowtie_{\mathcal{A}(\overline{b})} \text{sat}(S_1^{'} \sqcap_e S_1^{\mathcal{A}}, \mathcal{A}) \\
&\overset{8}{=} \text{sat}(S_1^{'} \sqcap_e S_1^{\mathcal{A}}, \mathcal{A}) \bowtie_{\mathcal{A}(\overline{b})} \text{sat}(S_2, \mathcal{A}) \\
&= \text{sat}(S_1, \mathcal{A}) \bowtie_{\mathcal{A}(b)} \text{sat}(S_2, \mathcal{A})
\end{aligned}$$

Equations 2 and 8 follow from the commutative properties of respectively $\sqcap$ and $\bowtie$ and equations 3 and 6 are their associative properties. With definition 25 we obtain equations 4 and 7 while 5 follows from the induction hypothesis since $\mathbb{n}(S_2 \sqcap_{\overline{b}} S_1^{'}) = k$.

Consider (ii):

$$\begin{aligned}
\mathrm{sat}(S_1 \sqcap_b S_2, \mathcal{A})) &= \mathrm{sat}(S_1 \sqcap_b (S_2^{'} \sqcap_e S_2^{\mathcal{A}}), \mathcal{A}) \\
&\overset{2}{=} \mathrm{sat}((S_1 \sqcap_b S_2^{'}) \sqcap_e S_2^{\mathcal{A}}, \mathcal{A}) \\
&\overset{3}{=} \mathrm{sat}(S_1 \sqcap_b S_2^{'}) \bowtie_{\mathcal{A}(e)} sat(S_2^{\mathcal{A}}, \mathcal{A}) \\
&\overset{4}{=} (\mathrm{sat}(S_1, \mathcal{A}) \bowtie_{\mathcal{A}(b)} \mathrm{sat}(S_2^{'}, \mathcal{A})) \bowtie_{\mathcal{A}(e)} \mathrm{sat}(S_2^{\mathcal{A}}, \mathcal{A}) \\
&\overset{5}{=} \mathrm{sat}(S_1, \mathcal{A}) \bowtie_{\mathcal{A}(b)} (\mathrm{sat}(S_2^{'}, \mathcal{A}) \bowtie_{\mathcal{A}(e)} \mathrm{sat}(S_2^{\mathcal{A}}, \mathcal{A})) \\
&\overset{6}{=} \mathrm{sat}(S_1, \mathcal{A}) \bowtie_{\mathcal{A}(b)} \mathrm{sat}(S_2^{'} \sqcap_e S_2^{\mathcal{A}}, \mathcal{A}) \\
&\overset{7}{=} \mathrm{sat}(S_1, \mathcal{A}) \bowtie_{\mathcal{A}(b)} \mathrm{sat}(S_2, \mathcal{A})
\end{aligned}$$

Equations 2 and 5 are the associative properties of respectively $\sqcap$ and $\bowtie$. Equations 3 and 6 follow by definition 25 and equation 4 is the induction hypothesis, $\mathbb{m}(S_1 \sqcap_b S_2^{'}) = k$.

■

The reason for modelling railway systems is to be able to simulate and analyse possible behaviours of the system and thus make improvements. In Chapter 3.2, we presented railroad components as Petri Net components with built-in safety. One can construct a railway layout based on these components, but there might be other properties about the system that one might want to explore that require a different Petri Net implementation of the railway components. This can include various aspects of collision detection, other kinds of train separation principles, other interlocking principles etc., or maybe using the same railroad components but exploring different layouts to achieve a better performance. By storing the specification as a separate data structure and using the saturation technique we decoupled an abstraction from its implementation so that the two can vary independently. The time spent designing and testing the Petri Net model is shortened since:

- It is more manageable to model railway systems at the specification level than at the Petri Net level. Especially for engineers unfamiliar with Petri Nets.
- With a specification, the underlying Petri Net implementation can be replaced by other implementations without changing the high level specification. Atomic saturation can be applied multiple times and by using a different set of Petri Net components each time, we achieve different implementations based on the same specification. This facilitates simulation and analysis

of different railway operation principles. For example, with the same railroad specification, we can generate one Petri Net model composed by safety components and one with collision detection components.

- Each component that is a part of a specification has a corresponding Petri Net component after an atomic saturation. If we remove a component, its corresponding Petri Net will also be removed and if we change the specification's composition then the underlying Petri net will also change its composition. This means that a specification can be modified without considering the underlying implementation as the underlying implementation will automatically comply with the specification. Typically, when doing capacity research, it is often necessary to modify the railroad layout to achieve a higher capacity.

In the next chapter, we will describe a prototype tool base on the predefined theory.

Chapter 5

Implementation of a Tool

To prove our concept, a prototype application, which we call the *RWS-Editor*, is implemented based on the algebra defined in Chapter 4. The application automatically generates an executable Petri Net from a specification. Since there are many existing tools that support the use of Petri Nets, the goal was not to develop another simulator, but to be able to interact with existing simulators, such as Design/CPN, in order to analyse Petri Net models.

In this chapter, the functionalities of the tool will be presented along with an outline of the implementation, is done in JAVA [2]. Since JAVA programming is outside the scope of this thesis, we will not go thoroughly into the implementation, but rather demonstrate the tool and its properties. This is done mostly through some examples of how railroad specifications are created in the tool and the Petri Nets that are generated automatically from these specifications. We are going to look into two cases of use of RWSEditor. One of them is a small sized railway circuit using only two different types of railroad components, presented in Section 5.3. Even though it is small, it is large enough to be the subject of later analysis. Another case is the Oslo subway, which will be presented in Chapter 6. Technical drawings were provided by Oslo Sporveier and using these, we are able to show how an industrial, real-sized and complex railroad system can be specified in the tool, thus helping us demonstrate the usefulness of our concept and tool.

5.1 Structure

The application consists of a Railway System specification editor and a generator. From a high level point of view, the generator takes a set of different Petri Net railway components specified in a Petri Net editor along with a specification and generates a Petri Net railway net based on these components. This net can then be loaded into a Petri Net simulator for further formal analysis. The file format used for Petri Net input and output is eXtensible Markup Language (XML) [3]. Since Design/CPN is one of the most elaborate Petri Net tools available, supporting both design and analysis of Coloured Petri Nets at the time being, the tool is designed to meet the DTD[1] (Document Type Definition) of Design/CPN's XML.

Figure 5.1 gives a high level representation of the data flow between Design/CPN and RWSEditor. The tool takes railway components as input through XML files and uses the DOM (Document Object Model) parser to give an internal representation of the elements [4]. A high level railway net is then created and saturated with these components. The saturated code is then translated to XML and then imported back into Design/CPN[2].

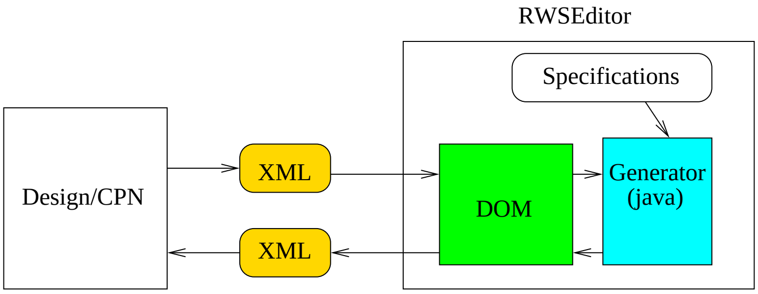


Figure 5.1: The data flow between RWSEditor and Design/CPN

5.2 Functionality

RWSEditor is a graphical tool. The tool provides functionality for:

---

[1] DTD defines the document structure with a list of legal elements.

[2] During the course of writing this thesis, Design/CPN was replaced by CPN-Tools. However, the XML format of Design/CPN may be converted to the XML format of CPN Tools.

- describing atomic specifications:
    - atomic components.
    - interface nodes and their types.
    - composition rules.
- constructing specifications by composition.
- saturation, both composite and atomic.
- loading XML files:
    - load Petri Net components from XML files.
    - load specifications from XML files.
- saving XML files:
    - save specifications to XML files.
    - save Petri Net implementations to XML files.

These functionalities will be described further. To avoid confusion with JAVA Nodes, we use the term connector when referring to interface nodes in the specification language.

5.2.1 Atomic Specification

The tool has eight built-in atomic components: end segment, turnout, rigid crossing, track segment, left and right singles, scissors and double slip. These are the most common components in railroad constructions and are used to form most railroad topologies [22]. Before beginning construction of a specification, some template atomic components must be created so that the component column is non-empty. These components have connectors (interface nodes) that connect to other components, and they work as templates for a specification. Initially, every connector of a template has the empty type Θ. All components are implemented as objects with unique identities and connectors are stored in an array in each component. The connectors are also objects and the component id and array index forms the unique identity for each connector.

To perform composition of components, rules have to be specified. This is done by first explicitly assigning each connector of a template with a type chosen from a list and then making rules based on these types.

Types are implemented as integers and more than one connector may have the same type.

It is also possible to have more than one copy of a given component as a template since we want to have the possibility to have the same component structure but with different Petri Net saturations later, e.g. two turnout templates saturated with different Petri Net implementations. To distinguish equal templates from each other, each component is equipped with an editable text label. When a component is created based on a template, it inherits the template's label, but this label can later be changed locally.

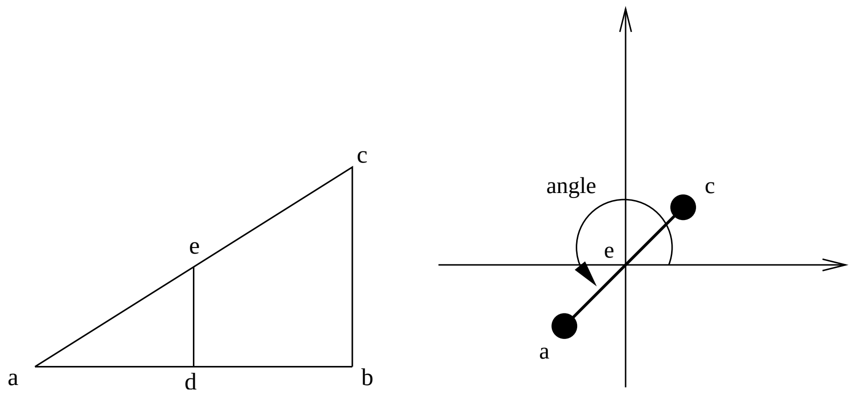
5.2.2 Specification

Each component in a specification is unique, and the constructed specification has a graph structure. Specifications are constructed by using the templates. When a template's connector is chosen, the template becomes the *selected template* and the chosen connector becomes the *selected connector*. When a selected connector is chosen (along with its selected template), any free connector in the specification can be chosen, and if the chosen connector is joinable with the selected connector, the specification is extended with a copy of the selected template. Two connectors are joinable if there is a rule with both their types. The new component will use its connector corresponding to the selected connector (belonging to the template component) to connect to the chosen connector. Internally, the joined connectors will refer to each other as neighbours. It is also possible to choose two free connectors of a specification and join them without using the templates, and to disconnect two joined connectors.

RWSEditor is implemented to ease the process of large scale Petri Net construction. Components can be rotated 360 degrees to form any wanted layout, and the structure of components can be adjusted by dragging their connectors. By far, the most used component is the road component, and by using the "create multiple nodes" option, a specified number of road components (if their template is joinable with itself) are automatically constructed and connected as if it was done manually step by step. This makes the construction process much more effective.

The Placement of components

The placement of each component is based on the coordinates of its connectors, which must be explicitly calculated. This is done by first calculating the coordinate of the center point of the component, which is the point e in Figure 5.2 (a). Thereafter, according to how many connectors this component has, an equal number of points evenly placed in a circle around e with a radius of ae, starting with the starting point a.



(a) (b)

Figure 5.2: The coordinate of components

Following is the pseudo code for calculating the coordinates of the connectors of a component:

```
calculatePositions () {

    <find the center point:>
    hypotenuse = $\sqrt{(ab^2 + bc^2)}$;
    angle a = $cos^{-1}$(ab/hypotenuse);
    radius = hypotenuse / 2;
    ad = radius * cos (angle a);
    de = radius * sin (angle a);
    <Calculate the positions of connectors:>
    make $e$ the origin;
    angle = angle between the x axis and ea; (see Figure 5.2 (b))
    angle between connectors = 360/number of connectors;

     <We have already the coordinates for start position, a. Must
    calculate the coordinates for the remaining connectors:>
```

```
        for (<all connectors $c$>){
        angle += angle between connectors;
        c.X = radius * cos(angle);
        c.Y = radius * sin(angle);
        }

}
```

5.2.3 Saturation

We may assign a Petri Net implementation to any template at any time during the construction of a specification. Since each component refers to its template, all components added to the specification will have the same underlying Petri Net implementation as their templates. The user can change the underlying Petri Net implementation of a template at run time.

Before we assign Petri Nets to templates, these nets must be loaded into RWSEditor as XML files. These files are parsed by Java library functions (DOM) and represented internally as objects of types Place, Transition and Arc. An assignment of a component to a Petri Net implementation is done by specifying the input file and explicitly assigning each connector of the component to an interface place.

5.2.4 The Petri Net Output

The Petri Net for the specification can be written to file according to the DTD of Design/CPN. The algorithm that builds the composite Petri Net will traverse components in a specification, which has an undirected graph structure, depth-first [33], and look at the underlying Petri Net. The algorithm uses variables to mark both the visited components and their visited connectors. Along the way, the algorithm will assign each Petri Net element with a unique id and a unique name for places and transitions. These names relay which node they corresponds to in the specification, more precisely, a concatenation of their original names and the components' IDs. The time to perform the traversal of a specification is $\mathcal{O}(|Connector| + |Component|)$ where $|Connector|$ and $|Component|$ are the number of connectors and components in a specification.

5.2.5 The Specification Output

The specification can also be saved as an XML file. Files contain the templates used, the constructed specification and the rules used. RWSEditor constructs XML files in accordance with the following DTD:

```
<!-- Project -->
<!ELEMENT rws (template, workspace, rules)?>

<!-- The template component -->
<!ELEMENT templates (node*)>

<!-- The workspace component -->
<!ELEMENT workspace (node*)>

<!-- Composition rules -->
<!ELEMENT rules (rule)*>

<!-- The node component -->
<!ELEMENT  node   (info, placement, endcoordinates?, connector*)>
<!ATTLIST  node   id                 ID     #REQUIRED
                  templref           IDREF  #IMPLIED>

<!-- Necessary information -->
<!ELEMENT  info       EMPTY>
<!ATTLIST  info       componenttype  CDATA  #IMPLIED
                      nodelength     CDATA  #REQUIRED
                      status         CDATA  #REQUIRED>

<!-- The coordinates and dimensions of the node -->
<!ELEMENT  placement  EMPTY>
<!ATTLIST  placement  x        CDATA  #REQUIRED
                      y        CDATA  #REQUIRED
                      width    CDATA  #REQUIRED
                      height   CDATA  #REQUIRED
                      centerX  CDATA  #REQUIRED
                      centerY  CDATA  #REQUIRED>

<!-- If this is an end element (only one connector) -->
<!ELEMENT  endcoordinates  EMPTY>
<!ATTLIST  endcoordinates  endp1x  CDATA  #REQUIRED
                           endp1y  CDATA  #REQUIRED
```

```
                           endp2x  CDATA  #REQUIRED
                           endp2y  CDATA  #REQUIRED>

<!-- The connector component -->
<!ELEMENT  connector       (pos, neighbour?, info)>
<!ATTLIST  connector       index       CDATA  #REQUIRED
                           noderef     IDREF  #REQUIRED
                           istemplate  CDATA  #REQUIRED>

<!-- Position -->
<!ELEMENT  pos    EMPTY>
<!ATTLIST  pos    x        CDATA  #REQUIRED
                  y        CDATA  #REQUIRED>

<!--  Neighbour -->
<!ELEMENT  neighbour    EMPTY>
<!ATTLIST  neighbour    node   IDREF  #REQUIRED
                        index  CDATA  #REQUIRED>

<!-- Vital information -->
<!ELEMENT  info    EMPTY>
<!ATTLIST  info    status        CDATA  #REQUIRED
                   connectortype CDATA  #REQUIRED>

<!-- Rule -->
<!ELEMENT  rule  EPTY>
<!ATTLIST  rule  from   CDATA  #REQUIRED
                 to     CDATA  #REQUIRED>
```

The DTD gives a description of data that constitute specifications. A node component is a basic atomic component in the specification. It consists of a unique id, a reference to its corresponding template, all the calculated coordinates and dimensions and its connectors, which each in turn has a position, a type and a reference to its joined connector, if any.

5.3 Cardamom Town Ride

We consider a simple Cardamom town's[3] railroad circuit consisting of four turnouts and represented as real railroad drawings, as illustrated in Figure 5.3.

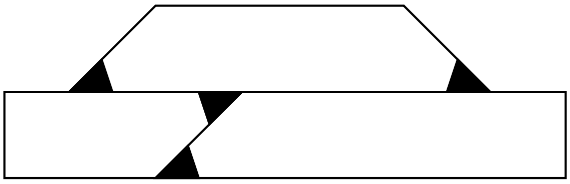


Figure 5.3: Cardamom circuit

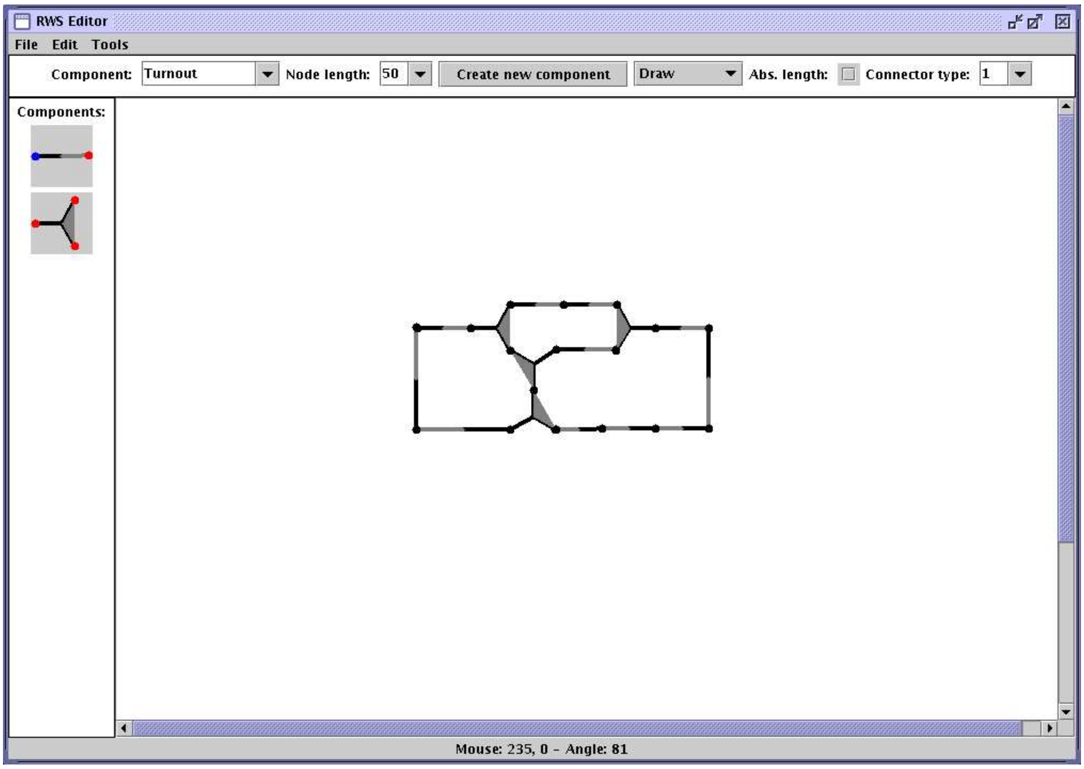
Figure 5.4 on the following page shows how the railroad circuit of Figure 5.3 is specified in RWSEditor, based on the atomic components denoted *Components*. The tool requires that the atomic specification is defined first, including the composition rules. In this example, the specification uses only two kinds of railroad components, track segments and turnouts.

The saturation algorithm, based on Lemma 7 and Theorem 1 on page 62 that automatically generates Petri Net code can be invoked after the atomic components are saturated with Petri Net turnout and road components, for example those presented in Section 3.2. Figure 5.5 shows how atomic saturation is done in the tool, by assigning each connector of the templates (the atomic components) to a concrete Petri Net interface place. Here, the blue connector represents the connector being assigned.

The Petri Net code, which is a concrete implementation of the specification, is produced in the form of an XML file, ready for input into Design/CPN. Figure 5.6 on page 77 shows the Petri Net automatically generated from the specification in Figure 5.4 in Design/CPN. In Chapter 7, the dynamic properties of this net will be analysed.

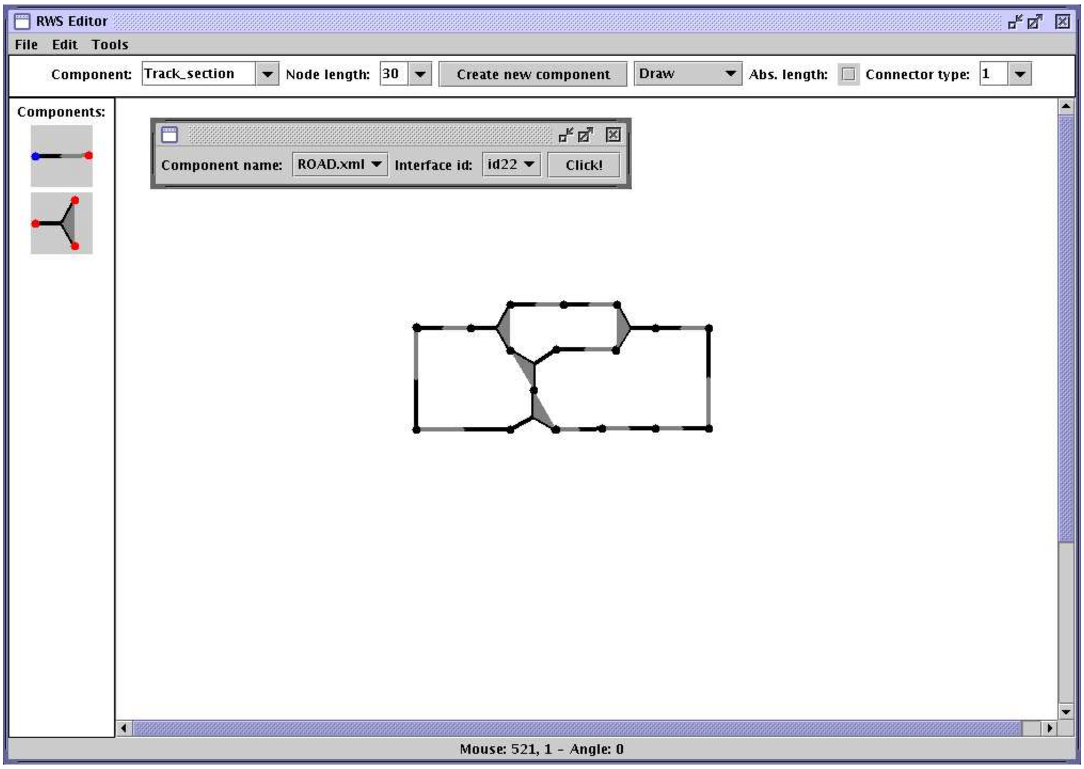
---

[3] Cardamom Town is a children's story, written by Thorbjørn Egner [27].



Component: Turnout
Node length: 50
Draw
Abs. length: ☐
Connector type: 1
Mouse: 235, 0 – Angle: 81

Figure 5.4: Cardamom circuit in RWSEditor



Component: Track_section
Node length: 30
Draw
Abs. length: ☐
Connector type: 1
Component name: ROAD.xml
Interface id: id22
Mouse: 521, 1 – Angle: 0

Figure 5.5: Saturation of the Cardamom circuit

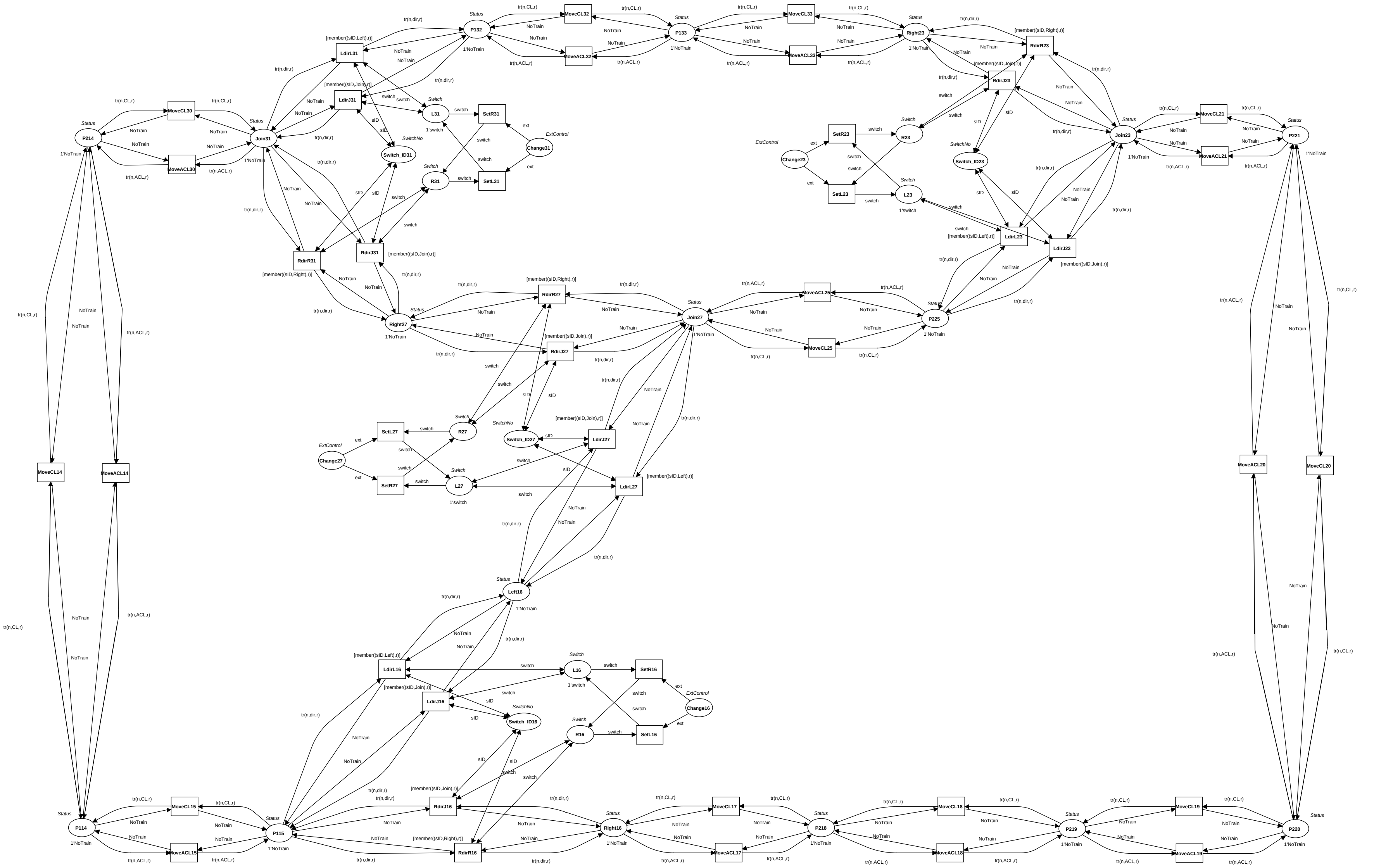


Figure 5.6: Petri Net model of the Cardamom circuit

Chapter 6

A Large Application — Oslo Subway

Oslo Sporveier [5] is the only public transportation company enclosed within the city limits of Oslo[1] and the only subway company in Norway. *Oslo subway* (Figure 6.1 on the following page) consists of 5 lines, operates in two main directions, east to west and west to east, and has a total of 103 stations. These 5 lines are:
line 1: Majorstuen - Frognerseteren - Majorstuen,
line 2: Ellingsrudåsen - Østerås - Ellingsrudåsen,
line 3: Mortensrud - Sognsvann - Mortensrud,
line 4: Bergkrystallen - Bekkestua - Bergkrystallen and
line 5: Vestli - Storo - Vestli.

The subway system is often undergoing changes and Oslo Sporveier is interested in integrating Petri Nets in the system development. The goal for Oslo Sporveier is to be able to simulate the Oslo subway and analyse schedules of trains, including the trains' routes through the network, arrival and departure time at stations, maximum speeds, etc. Being able to simulate changes in the infrastructure of the Oslo subway system would also be a great advantage. For example, Oslo subway is currently extending line number 5 from *Storo* station to *Sinsen* and then to *Carl Berner* station, forming a circuit.

Trains run on tracks that are divided into sections that at all time report where trains are located. Oslo subway also operates on a "fail safe" block system principle, which is based on train/no train in the

---

[1] There are other bus companies that have routes inside the city, but all these routes extend outside the city.

Figure 6.1: Oslo subway

sections (see Section 3.2.1). In practice, this is done by an electrical circuit with a source of current at one end and a detection device at the other. If a section is occupied by a vehicle, its axles produce a short circuit between the two rails. The detection device will not receive any current and therefore detects the section as occupied. Since our railway Petri Net components implement a block system, we may use them without modifications.

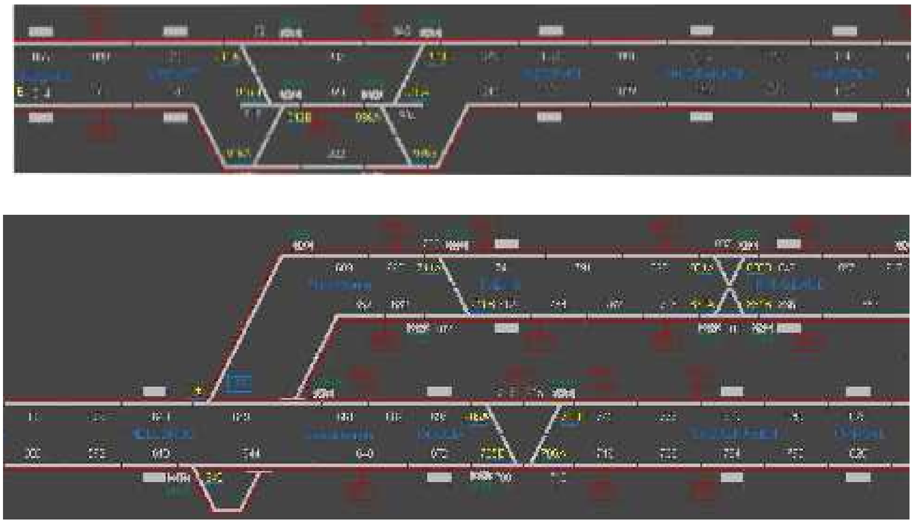
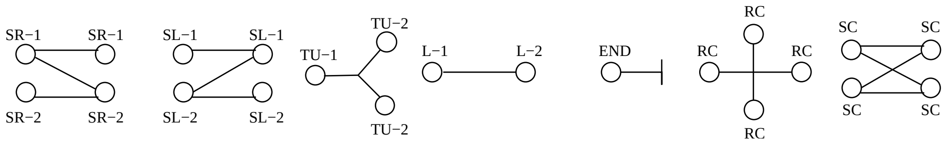


Figure 6.2: Oslo subway technical drawings

In cooperation with Oslo Sporveier, the whole subway system has been specified in the RWSEditor tool according to the technical drawings. The drawings where partially in electronic format and partially old-fashioned maps. Figure 6.2 depicts segments of the Oslo subway technical drawings. The topmost fragment is a part of line 5, between *Linderud* and *Ammerud* stations and the fragment in the botton is the crossroad between line 2 and 3, in the area between *Hellerud*, *Haugerud* and *Oppsal*. According to Oslo Sporveier, the drawings shall be interpreted as follows:

> **Tracks:** Divided into sections, each corresponding to three road components.
>
> **Scissors:** Shall be pairwise synchronised.

The whole of Oslo subway has been specified in RWSEditor using the composition rules in Figure 6.3. These rules are based on 11 interface types. It is hard to state absolute principles and rules for correct construction of railroads, so we have been pragmatic in following the most common patterns in the technical drawings.



<SR-2, SL-2>,	<SR-2, L-1>,	<SR-2, TU-2>,	<SR-2, SR-1>
<SR-2, SL-1>,	<SR-2, L-2>,	<SR-2, TU-1>,	<SL-2, L-1>
<SL-2, TU-2>,	<SL-2, SR-1>,	<SL-2, SL-1>,	<SL-2, L-2>
<SL-2, TU-1>,	<L-1, SC>,	<L-1, TU-2>,	<L-1, END>
<L-1, SR-1>,	<L-1, SL-1>,	<L-1, L-2>,	<L-1, RC>
<L-1, TU-1>,	<SC, L-2>,	<TU-2, SR-1>,	<TU-2, RC>
<TU-2, L-2>,	<TU-2, TU-1>,	<END, L-2>,	<SR-1, SL-1>
<SR-1, L-2>,	<SR-1, TU-1>,	<SL-1, L-2>,	<SL-1, TU-1>
<L-2, RC>,	<L-2, TU-1>,	<RC, RC>,	<RC, TU-1>
<TU-1, RC>,	<TU-1, TU-1>		

Figure 6.3: Composition rules for Oslo subway

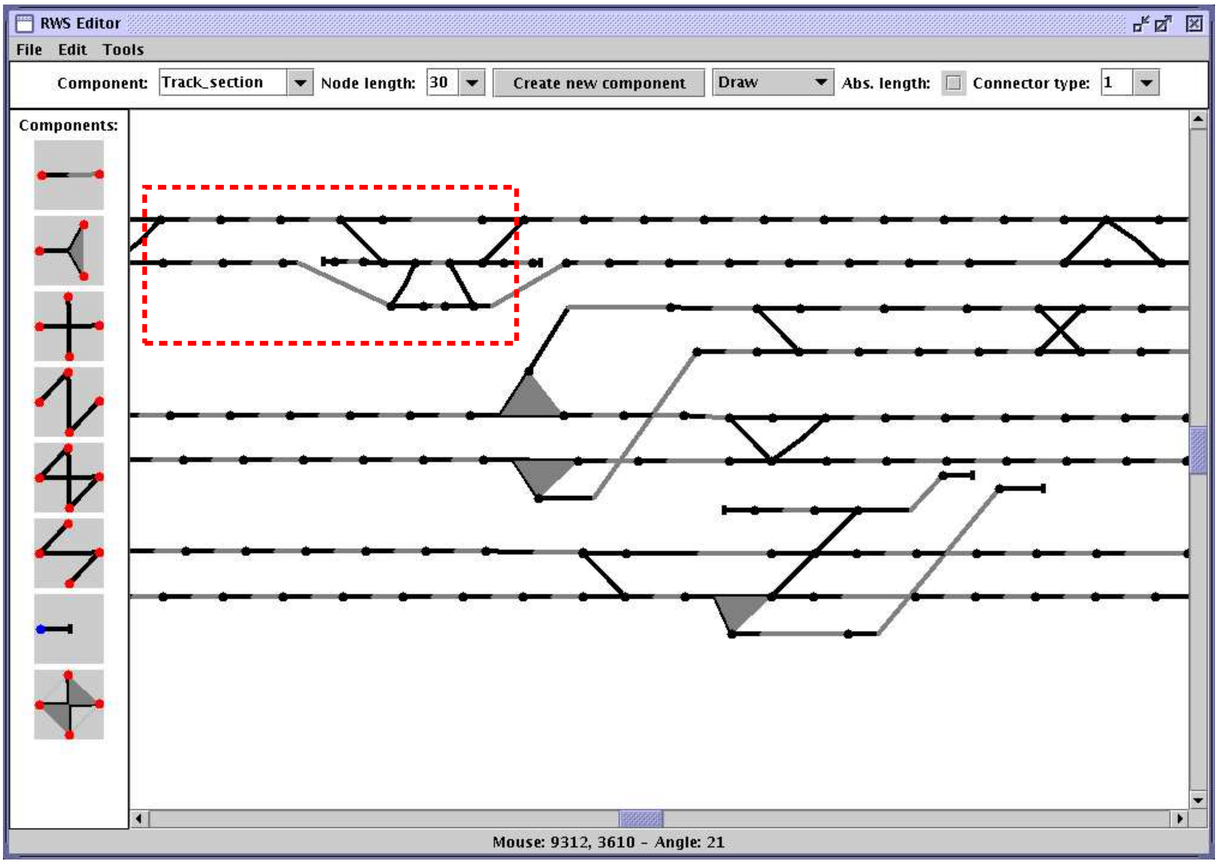
Take for instance the road component. It has two interface types, L-1 and L-2. These two types participate in rules with all other interface types except themselves, which means there are no rules $\langle L\text{-}1, L\text{-}1\rangle$ or $\langle L\text{-}2, L\text{-}2\rangle$. This is because the road components are directed, and to preserve the direction, these combinations can not be used. The end component has type END and $\langle L\text{-}1, END\rangle$ and $\langle END,\ L\text{-}2\rangle$ are the only rules with this type, to ensure that trains have sufficient room for deceleration and that they have sufficient room to properly leave any crossing areas.

During the construction of the Oslo subway specification, we had to interpret some components because the technical drawings are not completely consistent. In some places it is not clear whether a given component is a rigid crossing or a double slip, because the usage of indicators for points is ambiguous (some turnouts have a point indicator and others do not), and without the indicator, both components look the same. We solved this by simply using a rigid crossing if there were no point indicators. As for map drawings, it is not shown how many block sections there are between stations, so the number of block sections is determined by evaluating the distances and comparing these to drawings with block sections. Furthermore, the technical drawings do not provide details about the train stables so they were not modelled.

The specification process took approximately two working days. A fragment of the specification is shown in Figure 6.4 on the next page, which includes the track area shown by the technical drawings in Figure 6.2. During the generation of the Petri Net, Java ran out of memory, resulting in a segmentation fault. This was due to the high number of objects generated, and increasing the runtime memory pool for Java resolved this. The overall specification based on 8 atomic components consists of a total of 918 components, including:

- 752 track sections
- 38 turnouts
- 4 rigid crossing
- 33 single left
- 44 single right
- 9 scissors
- 38 end sections
- a total of 2016 connectors

With the specification of Oslo subway and the basic railway components given in Chapter 3.2, RWSEditor automatically generated the Petri Net



Component: Track_section
Node length: 30
Draw
Abs. length: ☐
Connector type: 1
Mouse: 9312, 3610 - Angle: 21

Figure 6.4: A fragment of the Oslo subway specification

implementation. Figure 6.5 on page 85 shows a small fragment of the generated Petri Net imported into Design/CPN, corresponding roughly to the framed part of the upper tracks in Figure 6.4. To summarise, the generated Petri Net has 33031 Petri Net elements, including 3455 places, 5726 transitions, 23850 arcs and 2753 initial tokens.

We found out that Design/CPN[2] was not able to handle Petri Nets the size of the generated Petri Net implementation (Figure 6.5 filled the whole working space of Design/CPN in its width), so we could not perform neither analysis nor simulations on the net. Theoretically speaking, given that Oslo subway is one bounded, each place contains maximum one token at any time. This means that the occurrence graph has a maximum of 2^n reachable states, where n is the number of places and the number of states increases exponentially with the

---

[2] We have also tried cpnTools, it succeed in processing the file but because of its size, it is impossible to work with on the hardware we had access to.

number of trains. Even though real-sized railway systems have many constraints that reduces the number of states a system can be in, e.g. by trains following concrete routes and operational rules, calculating a full occurrence graph still requires a vast amount of time and memory.

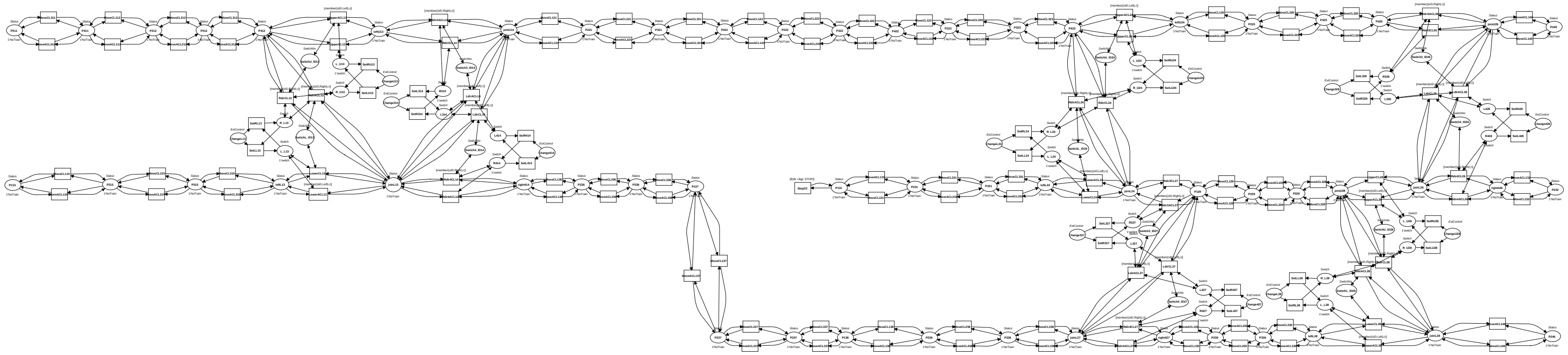


Figure 6.5: A fragment of the Oslo subway Petri Net model

Chapter 7

Analysis

This chapter illustrates how properties that may be interesting in the domain of railway systems can be analysed. We look at the Cardamom net that we constructed earlier using RWSEditor.

7.1 The Railway Domain

Some properties associated with railway systems can be reduced to properties associated with Petri Nets. Safety, in terms of keeping trains from colliding, is an important property associated with railway systems. To verify that a railway system is safe in this sense, we may examine the tokens in all places that represent track sections in the corresponding Petri Net. If we let (N, M_0) be a railway net N with initial marking M_0, P_t be a finite set of places that represents track sections in N and T be a function that returns the number of train tokens in a multi-set, then a Petri Net satisfying the safety property can be expressed as:

$$\forall M \in M_0^R, s \in P_t \ : \ T(M(s)) = 0 \vee 1$$

The formula expresses that each place in every reachable marking from the initial marking has either no train or exactly one train.

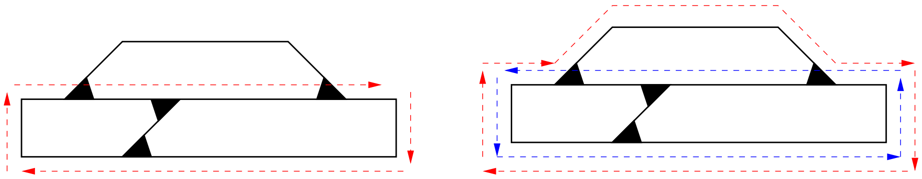
It is often interesting to see whether a system is in progression. The progression property indicates that the system is in a state where at least one train is able to move. One way to find out whether a railway system satisfies the progression property is to investigate whether at least one train can change its current position, to the track section ahead or behind. In a Petri Net this can be done by investigating transitions that move trains forward, more specifically, the transition firings. This property can be expressed by:

$$\forall M \; \exists M^{'}, t \in T_t, \; \sigma : M \xrightarrow{\sigma} M^{'} \wedge t \in \sigma$$

where $M, M' \in M_0^R$ and T_t is a finite set of transitions that are responsible for moving trains.

7.2 Analysis of the Cardamom Town Railway Net

We shall take the Petri Net model of the Cardamom town railway net generated in Section 5.3 and analyse its properties using simulations and State space methods. We attempt to analyse two initial states of the Cardamom net, one where all trains are running in the same direction along the same route (Figure 7.1 (a)) and another where trains are running along different routes and in different directions (Figure 7.1 (b)). In both cases, the starting point for trains remains the same.


(a) Case 1

(b) Case 2

Figure 7.1: Two analysis cases

7.2.1 Analysis of Initial State 1

We consider an initial state, the initial marking in Figure 7.2 on page 93, with two trains,
tr((3,CL,[(1,Right),(2,Join),(3,Join),(4,Join)])) and
tr((5,CL,[(1,Right),(2,Join),(3,Join),(4,Join)]))
running along the same route. The two red tokens are trains and the position of each turnout is indicated by the placement of a blue token. The grey tokens are NoTrain tokens and tokens with red borders are the identities of turnouts. The enabled transitions in this initial marking are marked with green borders.

By invoking the simulation option in Design/CPN, we may see one possible run with tokens moving between places and the enabled transitions. The first six simulation steps are shown below, the increasing leftmost numbers indicate the step number followed by the transition that fired. On the line that follows is the binding element, where n is the train line (of type *TrainLineNo*) and r is the route (of type *ListRoute*):

```
1  MoveCL14

    {n = 5, r = [(1,Right),(2,Join),(3,Join),(4,Join)]}

2  MoveCL30

    {n = 5, r = [(1,Right),(2,Join),(3,Join),(4,Join)]}

3  MoveCL20

    {n = 3, r = [(1,Right),(2,Join),(3,Join),(4,Join)]}

3  RdirR31

    {dir = CL, n = 5, r = [(1,Right),(2,Join),(3,Join),(4,Join)],
     sID = 1}

4  RdirJ27

    {dir = CL, n = 5, r = [(1,Right),(2,Join),(3,Join),(4,Join)],
     sID = 3}

5  MoveCL25

    {n = 5, r = [(1,Right),(2,Join),(3,Join),(4,Join)]}

6  LdirJ23

    {dir = CL, n = 5, r = [(1,Right),(2,Join),(3,Join),(4,Join)],
     sID = 2}

6  MoveCL19

    {n = 3, r = [(1,Right),(2,Join),(3,Join),(4,Join)]}
```

The simulation is based on a non-deterministic choice of enabled transitions. As we can see, at the beginning, train *5* will run first, leaving two track sections behind while train *3* stays in the same place. Then both trains will move concurrently for one step, after which train *3* will stop again while train 5 moves on. As we have the disadvantage of not having a real time concept represented in our model, we can not tell how much train *3* is behind schedule, we may only say how many steps or how many track sections.

With non-deterministic behaviour, we are not interested in the end marking, but rather the dynamic behaviour of the system, the possible markings. Here it may be interesting to see whether there are markings where trains may crash, if all trains are in progression or if we can achieve a *deadlock*. By deadlock, we mean a marking where all trains are stuck and no transition is enabled, violating the progression property. Here we will try to search for such a marking and whether the net is safe using state space analysis. The state space analysis calculated the occurrence graph for this initial marking, the graph has 156 nodes and 286 arcs. Some of the possible behaviour is summarised below.

Safety

The upper and lower integer bounds for each place in the net is calculated. Design/CPN calculates the upper and lower bounds using a function F of type *Node* → *multi-set* and calculates an integer $|F(n)|$ for each node n in the occurrence graph, returning respectively the maximum and minimum of the calculated integers.

Each line below (a total of 33 lines) has three attributes, corresponding to a place in the Cardamom circuit, the upper integer bound for that place and the lower integer bound for that place. *CardamomPetriNet'place_name* denotes the place with name *place_name* in the net named *CardamomPetriNet*. Lines 1 to 4 represent the *Change* place of each turnout, lines 8 to 11 are their corresponding left positions and lines 23 to 26 are their right positions. Places that hold each turnout's ID are in lines 30 to 33. The rest of the lines are the segment places.

```
       Boundedness Properties
      --------------------------------------------------
       Best Integers Bounds           Upper   Lower
1      CardamomPetriNet'Change16      0          0
2      CardamomPetriNet'Change23      0          0
3      CardamomPetriNet'Change27      0          0
4      CardamomPetriNet'Change31      0          0
5      CardamomPetriNet'Join23        1          1
6      CardamomPetriNet'Join27        1          1
7      CardamomPetriNet'Join31        1          1
8      CardamomPetriNet'L16           0          0
9      CardamomPetriNet'L23           1          1
10     CardamomPetriNet'L27           0          0
11     CardamomPetriNet'L31           0          0
12     CardamomPetriNet'Left16        1          1
13     CardamomPetriNet'P114          1          1
14     CardamomPetriNet'P115          1          1
15     CardamomPetriNet'P132          1          1
16     CardamomPetriNet'P133          1          1
17     CardamomPetriNet'P214          1          1
18     CardamomPetriNet'P218          1          1
19     CardamomPetriNet'P219          1          1
20     CardamomPetriNet'P220          1          1
21     CardamomPetriNet'P221          1          1
22     CardamomPetriNet'P225          1          1
23     CardamomPetriNet'R16           1          1
24     CardamomPetriNet'R23           0          0
25     CardamomPetriNet'R27           1          1
26     CardamomPetriNet'R31           1          1
27     CardamomPetriNet'Right16       1          1
28     CardamomPetriNet'Right23       1          1
29     CardamomPetriNet'Right27       1          1
30     CardamomPetriNet'Switch_ID16   1          1
31     CardamomPetriNet'Switch_ID23   1          1
32     CardamomPetriNet'Switch_ID27   1          1
33     CardamomPetriNet'Switch_ID31   1          1
```

As expected, since all railway components used in the saturation process (when we specified the net in Section 5.3) are safety components from Section 3.2, the upper and lower bounds of all track sections are one. Either there is a train (Train token) in the section or not (noTrain token). There has been no changes in the positions of any turnout as both the upper and lower bounds of all the *Change* places are 0, thus the controlling branch of all point machines stay the same throughout all markings of the net.

Liveness property

To find possible deadlocks, the Design/CPN function *ListDeadMarkings()* will search the entire occurrence graph, trying to find nodes that have empty lists of output arcs. Our search results:

```
ListDeadMarkings();
val it = [] Node list
```

As the function returns an empty list of dead markings, there are no deadlocks. This also means that the system satisfies the progression property.

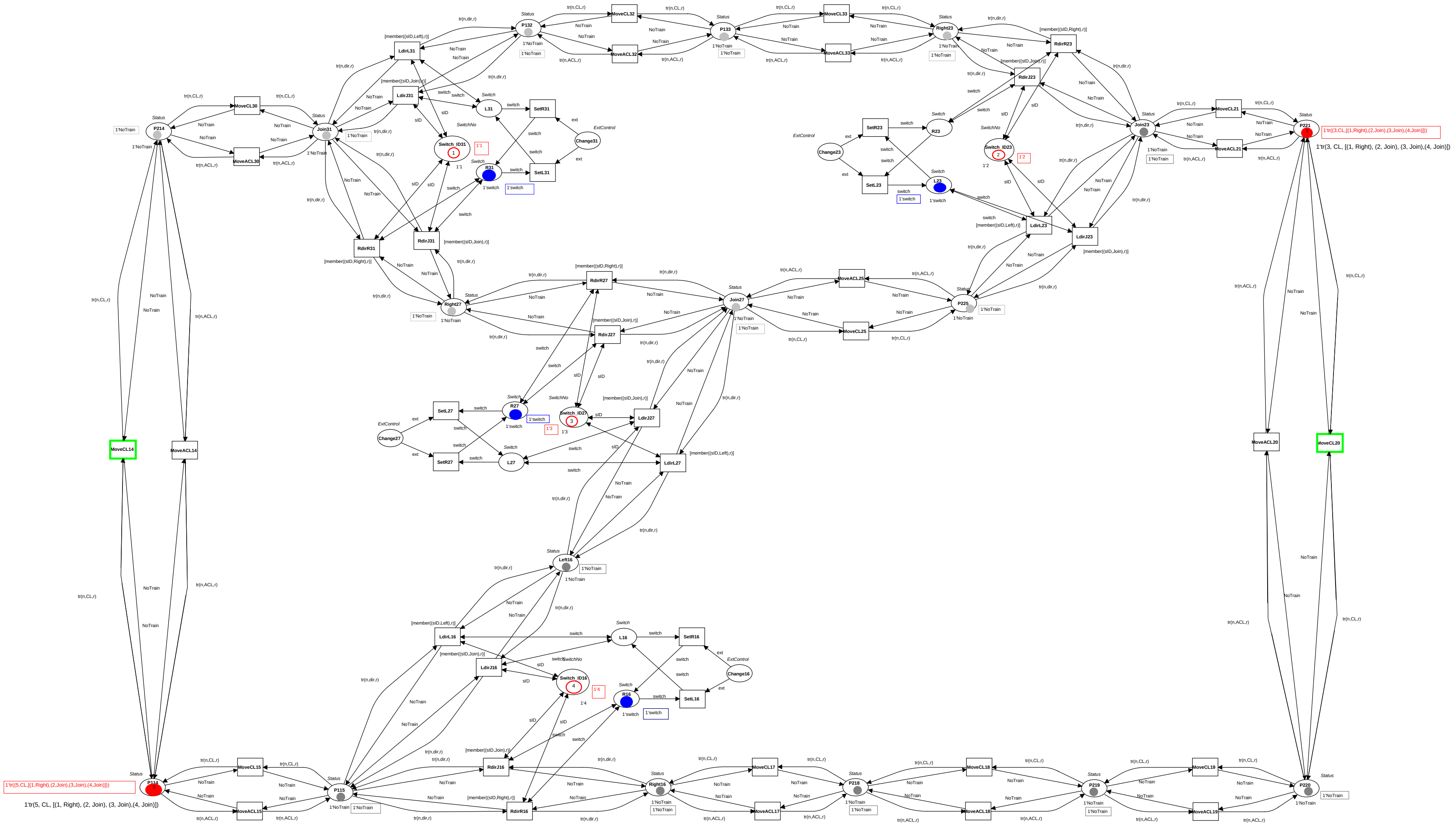


Figure 7.2: The Cardamom Petri Net

7.2.2 Analysis of Initial State 2

We now consider the same initial state of the net as in Figure 7.2 on the page before with the same markings, but with the trains running in opposite directions.
tr((3,ACL,[(1,Join),(2,Left),(3,Right),(4,Right)])) and
tr((5,CL,[(1,Left),(2,Join),(4,Join)]))
running in opposite directions and following different routes. Both trains reside in the same places as in case 1.

Liveness property

The occurrence graph for this marking contains several possible deadlocks, each represented by a marking with trains located on the conflicting route, which is the route where the travel plans of both trains overlap (see Figure 7.1 (b)).

There are different ways to avoid conflicts regarding the initial marking. One is to design road components to turn the train when it meets an opposing train, a train in the opposite direction, e.g. from going clockwise to anti-clockwise. Another is to wait for clearance of the conflicting route before entering it by checking the states of track sections on that route. The first approach is used in [6], where trains do not follow any concrete travel plan except for the direction. The disadvantage of this approach is that when two trains with different directions meet on a conflicting route, they will both change their direction and start to move away from each other, resulting in an unrealistic schedule and in the worst case, repeating this pattern indefinitely, which will be the case with our net. With the second approach, each train has to wait until the train in the conflicting area has left. The number of arcs needed to do the check increases proportionally with the number of sections in the conflicting area. Using the second approach, the calculated occurrence graph has a total of 228 nodes and 427 arcs with no deadlocks. By forcing the simulator to fire all enabled transitions in each marking we can observe the behaviour of trains in a run where trains compete to enter the conflicting area:

```
6  MoveCL33
    {n = 5, r = [(1,Left),(2,Join),(4,Join)]}

6  RdirJ31
    {dir = ACL, n = 3, r = [(1,Join),(2,Left),(3,Right),(4,Right)],
```

```
    sID = 1}

7  MoveACL30
    {n = 3, r = [(1,Join),(2,Left),(3,Right),(4,Right)]}

7  SetR23
    {}

8  MoveACL14
    {n = 3, r = [(1,Join),(2,Left),(3,Right),(4,Right)]}

9  MoveACL15
    {n = 3, r = [(1,Join),(2,Left),(3,Right),(4,Right)]}

10 RdirR16
    {dir = ACL, n = 3, r = [(1,Join),(2,Left),(3,Right),(4,Right)],
    sID = 4}

11 MoveACL17
    {n = 3, r = [(1,Join),(2,Left),(3,Right),(4,Right)]}

                ⋮

17 LdirL23
    {dir = ACL, n = 3, r = [(1,Join),(2,Left),(3,Right),(4,Right)],
    sID = 2}

18 MoveACL25
    {n = 3, r = [(1,Join),(2,Left),(3,Right),(4,Right)]}

18 SetR23
    {}

19 RdirJ23
    {dir = CL, n = 5, r = [(1,Left),(2,Join),(4,Join)], sID = 2}

19 RdirR27
    {dir = ACL, n = 3, r = [(1,Join),(2,Left),(3,Right),(4,Right)],
    sID = 3}

20 MoveCL21
    {n = 5, r = [(1,Left),(2,Join),(4,Join)]}
```

```
21 MoveCL20
    {n = 5, r = [(1,Left),(2,Join),(4,Join)]}
```

In step 6, both trains moved concurrently forward, but in this step train 3 entered the conflicting area. In step 7, Train 3 continued moving while train 5 stayed in the same track section, waiting for the position of turnout 2 to change so that it could enter the conflicting area. In the steps from 8 to 17 only train 3 could move since it was still in the conflicting area and train 5 still was waiting for the clearance. It was not until step 18 that train 3 left the conflicting area, giving train 5 clearance to enter (step 19) while train 3 this time had to wait in the entrance to the area.

Chapter 8

Conclusion

In this thesis, we have seen how railway components can be modelled using Coloured Petri Nets, formally defined a way to automatically generate Petri Nets and implemented a tool that does this in JAVA. We shall conclude this thesis by reconsidering and answering the questions posed in the introduction. These questions comprise using Coloured Petri Nets for railway modelling, automatic construction and analysis.

Railway Models as Coloured Petri Nets

> How can we use Coloured Petri Nets to model railway components naturally with concrete operational rules and trains?

We illustrated the usability of Coloured Petri Nets to model railway systems, both the basic elements in a railway system, such as track sections, trains etc., and behaviours like for example trains movements.

We showed how track sections can be represented naturally by places and trains by structured tokens. Since Coloured Petri Nets are very general, they can be used to model the different basic railway components, such as road segments, turnouts, crossings, double slips, rigid crossings, scissors and singles. These components can in turn be composed to form different railroad topologies. Furthermore, operations that control the routing of trains can also be expressed. We have seen how point machines explicitly can be modelled and together with guards they can control the routing of tokens. These Coloured Petri Net railway components implement the basal aspects of a block system operation to ensure a safe train separation.

A Coloured Petri Net model can be used both to describe the states of a system and the actions that alter these states. For railway systems, the state of a system can be represented by a given distribution of trains in track sections. In an analogous way, the distribution of tokens over the places defines the state of a system. That trains may enter or leave track sections are behaviours of a railway system. This corresponds to firings of transitions which moves tokens from places to places, producing changes in the distribution of tokens.

Railway systems are concurrent systems and coloured Petri Nets is adequate for expressing arrangements associated with concurrent systems such as concurrency, sequencing and choice. Properties such as allowing multiple trains to concurrently run on tracks and sequential train movement can easily be expressed by the few and simple mathematical entities of Coloured Petri Nets. Another aspect worth noticing about the benefits of using Coloured Petri Nets for railway modelling is the possibility of modular composition and progressive modelling. For example, we may construct a railway net by composition of the basic railway components as we have done and these Coloured Petri Net components can be refined with additional properties, for example synchronisations as we have seen in the scissors components.

For the above reasons, we think Coloured Petri Nets is adequate and well suited for modelling real life systems such as railway systems.

Automatic Construction

> How can we automatically construct Petri Net models?
> What kind of algebra is sufficient for this construction?
> What are the benefits of this construction if any?

We introduced an abstraction by using a two layered model for automatic construction of complex Petri Nets. The theory introduces a specification language with structure and rules for creating railway specifications and a saturation technique. The specifications are constructed through recursive composition, and we built a specification out of basic components. Saturation works as a bridge between these two layers and is the essence of automatic construction, as it takes an instance of the specification layer, a concrete specification and generates an instance of the Petri Net layer, a concrete Petri Net model.

We introduced an algebra for composition and decomposition for both the specifications and the Petri Nets. Compositions are necessary for building the specifications and the Petri Nets and decompositions to split or remove subsets from the specifications and the Petri Nets.

With the approach described in Chapter 4, constructing Petri Nets by specifications and using the saturation technique have some advantages:

First, it is more manageable to model railway systems at the specification layer than at the Petri Net layer. In the examples of Chapter 5, we have seen railway specifications in a style that closely resembles technical railway drawings, so no particular Petri Net knowledge is needed and we do not need to worry about whether placements of places, transitions and arcs are correct.

Second, the time spent constructing the specification is considerably shorter than it would be had the specification been constructed using Petri Nets. Take for example Oslo subway, an industrial sized system, where it took two days to specify and generate an executable net using RWSEditor. This system would perhaps require weeks or months if it was modelled directly in Petri Nets.

Third, the structure of the layout is stored, which makes it easy to change the underlying implementation as the underlying Petri Net components can be replaced without altering the high level specification. By saturating the same specification with different Petri Net components, a set of executable Petri Net codes can be generated. This is an effective way to simulate and analyse different railway operation principles.

Fourth, the specification can be modified without considering the underlying implementation. As we have seen in Sections 4.1 and 4.2, composition is defined for both the specification and the Petri Net layer. If the structure in the specification is changed, the underlying Petri Net implementation will automatically be modified.

Fifth, it is possible to extend the abstraction layer and Petri Net implementation layer independently. We may for example add new types of railway components in the specification layer or even use a composite specification as a basic building block without considering how this is done in the Petri Net layer.

For these reasons, this way of constructing railway systems facilitates experimentation with railway structures and behaviours. The techniques presented in this thesis generalise to many application domains and not just railway systems. Domains with many instances of components and non-trivial but formalised grammars for connecting components seem particularly suitable for saturations.

Analysis

What are the benefits of analysis methods provided by
Petri Nets, when applied to railway systems?

We have seen how occurrence graphs can be used to verify dynamic properties of railway systems from an initial state and how simulations can be used to observe behaviours of the system. With simulations we may see what is actually happening in the system, where the trains are, if they operates correctly etc.

If the net is bounded, state space analysis provides a complete knowledge of all its properties, because then the occurrence graph is finite. This is the case with railway nets constructed with the defined components — we have a finite number of initial tokens, including trains, and all places in the Petri Nets constructed by these components are bounded. This means that there is a finite number of states to consider when analysing the behaviours. For analysing small nets, as in Chapter 7.2, the occurrence graph is fairly small, but for Oslo subway, we may see an exponential growth in size. Thus, for large Petri Nets such as the Oslo subway model, sufficient time and memory are necessary along with techniques to cut down the search space of an occurrence graph.

8.1 Future Work

There are several directions for future work in the railway domain and some of them have already started.

Modelling railway systems.

This thesis has only shown how Petri Nets can be used to implement the basic railroad components and operations that comprise parts of an industrial sized railway system, while signalling systems, control systems, interlocking systems, stations etc. also all constitute important parts. Furthermore, the concept of time in this thesis correspond to the notion of steps in the firing semantics of Petri Nets, which is implicitly given. A more detailed modelling will require time being modelled explicitly, for example by using timed Coloured Petri Nets [15], so that the ideas of timetables, durations and delays can be implemented. A notion of time is also necessary to be able to analyse the performance of systems .

Domain specific analysis and complexity

Domain specific analysis methods are specialised used to solve problems in a well-defined application domain.

Complexity of general Petri Net problems has been studied in many papers. As shown in [10, 9, 32, 28], most interesting questions (e.g. liveness and boundedness) about the behaviour of general Petri Nets are EXPSPACE hard [11]. For some restricted Petri Net classes, these problems are tractable. For railway systems, we are curious about whether railway nets by reduction can be shown to belong to a restricted class of Petri Nets. An important question to answer is where problems regarding the behaviour of railway nets belong on the complexity map.

For this purpose, it is necessary to investigate generalisations of domain specific analysis, which is to see whether we can use the advantage of considering one particular domain, i.e. the railway domain, and its components, to achieve better computability analysis. This approach was taken by Wil van der Aalst in his dissertation where he showed how Petri Nets can be used to define, analyse and implement the concept of both logistics and workflow [29, 31, 30]. He proved for example the relation between Free Choice Petri Nets [8] and workflow nets.

Bibliography

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

[2] http://java.sun.com/.

[3] http://www.w3.org/XML/.

[4] http://www.w3.org/DOM/.

[5] http://www.sporveien.no/.

[6] G. Berthelot and L. Petrucci. Specification and validation of a concurrent system: An educational project. In K. Jensen, editor, *DAIMI PB: Workshop Proceedings Practical Use of High-level Petri Nets*, pages 55–72. University of Aarhus, Department of Computer Science, jun 2000.

[7] Eike Best, Raymond Devillers, and Maciej Koutny. *Petri Net Algebra.* Springer-Verlag, 2001.

[8] Jörg Desel and Javier Esparza. *Free Choice Petri nets.* Number 40. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1995.

[9] Javier Esparza. Decidability and complexity of petri net problems - an introduction. In *Lectures on Petri Nets I: Basic Models. Advances in Petri Nets*, number 1491 in Lecture Notes in Computer Science, pages 374–428. Springer-Verlag, 1998.

[10] Javier Esparza and Mogens Nielsen. Decidability issues for petri nets. *Petri Net Newsletter*, (47):5–23, 1994.

[11] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1990.

[12] Kurt Jensen. *Coloured Petri Nets - Basic Concepts, Analysis Methods and Practical Use*, volume 3 of *EATCS, Monographs on Theoretical Computer Science*. Springer-Verlag, 1997.

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

[14] Kurt 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.

[15] Kurt Jensen. An Introduction to the Practical Use of Coloured Petri Nets. Obtained from `http://www.daimi.aau.dk/PetriNets/`, 2002.

[16] Kurt Jensen. An Introduction to the Theoretical Aspects of Coloured Petri Nets. Obtained from http://www.daimi.aau.dk/PetriNets/, 2002.

[17] Kurt Jensen. A Short Introduction to Coloured Petri Nets. Obtained from http://www.daimi.aau.dk/PetriNets/, 2002.

[18] Thor Kristoffersen, Anders Moen, and Hallstein Asheim Hansen. Extracting High-Level Information from Petri Nets: A Railroad Case. *Proceedings of the Estonian Academy of Physics and Mathematics*, 52(4), December 2003.

[19] Meta Software Corporation, Cambridge, MA U.S.A. *Design/CPN Reference Manual*.

[20] Meta Software Corporation, Cambridge, MA U.S.A. *Design/CPN Tutorial*.

[21] Anders Moen and Ingrid Chieh Yu. Large scale construction of railroad models from specifications. *IEEE SMC, Systems, Man and Cybernetics*, 10 2004.

[22] Joern Pachl. *Railway Operation and Control*. VTD Rail Publishing, 2002.

[23] Carl Adam Petri. Kommunikation mit Automaten. Technical Report Schriften des IIM Nr. 2, Bonn: Institut für Instrumentelle Mathematik, 1962.

[24] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[25] W. Reisig and G. Rozenberg, editors. *Lectures on Petri Nets II: Applications*, volume 1492 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1998.

[26] Wolfgang Reisig. *Petri Nets, An Introduction*, volume 2 of *EATCS, Monographs on Theoretical Computer Science*. Springer-Verlag, 1985.

[27] Egner Torbjørn. *When the Robbers came to Cardamom Town (English edition)*. Cappelen, 1993.

[28] Antti Valmari. The State Explosion Problem. In Reisig and Rozenberg [24].

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

[30] Willibrordus Martinus Pancratius van der Aalst. Three Good Reasons for Using a Petrinet based Workflow Management System. In T. Wakayama, S. Kannapan, C.M. Khoong, S. Navathe, and J. Yates, editors, *Information and Process Integration in Enterprises: Rethinking Documents*. Kluwer Academic Publishers, Boston, Massachusetts, 1998.

[31] Willibrordus Martinus Pancratius van der Aalst, K. M. van Hee, and G. J. Houben. Modelling and Analyzing Workflow using a Petrinet based Approach. In G. De Michelis, C. Ellis, and G. Memmi, editors, *Proceedings of the second Workshop on Computer-Supported Cooperative Work, Petri nets and related formalisms*, pages 31–50, 1994.

[32] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. *Lecture Notes in Computer Science*, 1043:238–266, 1996.

[33] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java*. Addison Wesley, 1999.

[34] Ingrid Chieh Yu and Anders Moen. From modeling to analysis of railway systems using coloured petri nets. In *Proceedings of the 15th Nordic Workshop on Programming Theory (NWPT)*, 2003.

Appendix

Work on this thesis has resulted in an executable application RWSEditor, for specifying and automatically constructing large Petri Net models of railroads. The appendix presents the JAVA code for RWSEditor.

JAVA code

Listing 1: RWSEditor.java

```java

/**
 * Topmost class. This class includes the main () method and initiates
 * everything. No particular other functions.
 */
class RWSEditor {

    static boolean DEBUG = false;
    static XMLUtils xmlUtils;
    static RWSEditorFrame frame;

    public static void main (String [] args) {
        frame = new RWSEditorFrame ();
        xmlUtils = new XMLUtils ();
    }
}
```

Listing 2: RWSEditorFrame.java

```java
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
import java.util.HashMap;
import java.util.Vector;
import java.util.Enumeration;
import java.awt.Graphics2D;
import java.awt.BasicStroke;

import java.beans.XMLEncoder;
import java.beans.XMLDecoder;
import java.beans.ExceptionListener;
import java.io.*;

/**
 * The RWSEditorFrame holds the utilities for RWSEditor.
 */
public class RWSEditorFrame implements MouseListener,
                                       KeyListener,
                                       ActionListener,
                                       ItemListener,
                                       MouseMotionListener {
    /* Panel and labels */
    JLabel statusbar;
    JPanel toolbar;
```

```
    JPanel templatebar;
    JFrame frame;
    JFileChooser chooser;
    BackgroundPanel panel;

    private int WIDTH = 900;
    private int HEIGHT = 600;
    private HashMap menuMap;
    protected int startX, startY;
    private Vector rwsNodes;
    private Vector rwsNodeTemplates;
    public static Insets insets;
    private static RWSNode currentNodeTemplate;
    private boolean createNodeTemplate;

    protected final static int DRAWING = 0;
    protected final static int CREATE_RULES = 1;
    protected static int action;
    protected static JPopupMenu popup, nodePopup;
    protected static Container bg;
    protected boolean alwaysAbsolute = false;

    private Dimension size;
    JScrollPane scroller;

    protected static int global_largestX, global_largestY = 0;
    protected static int global_smallestX, global_smallestY = 0;
    protected static int meanX, meanY = 0;

    int currentNumberOfConnectors = 2;
    String currentComponent = "Track_section";

    static boolean justTheLine = false;
    static int mouseX, mouseY;

    /* Constuctor */
    public RWSEditorFrame () {
        JFrame.setDefaultLookAndFeelDecorated (true);

        frame = new JFrame ("RWSEditor");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        bg = frame.getContentPane();

        JMenuBar menuBar = new JMenuBar();
        menuBar.setOpaque(true);
        menuBar.setPreferredSize(new Dimension(WIDTH, 20));

        JMenu m;
        JMenuItem item;
        menuMap = new HashMap(50);

        /* Do something with this menu */
        Menu [] menu = getMenu();
        for(int i=0;i<menu.length;i++){
```

```
                if(menu[i] == null)
                    continue;
                m = new JMenu(menu[i].getText());
                menuBar.add(m);
                for(int j=0;j<menu[i].items.length;j++){
                    if(menu[i].items[j] == null)
                        continue;
                    item = new JMenuItem(menu[i].items[j].getText());
                    item.addActionListener(this);
                    if(menu[i].items[j].hasSC()){
                        if(menu [i].items [j].hasModifier ())
                            item.setAccelerator(
                                KeyStroke.getKeyStroke(
                                    menu [i].items [j].getSC (),
                                    menu [i].items [j].getModifier ()));
                        else
                            item.setMnemonic(menu[i].items[j].getSC());
                    }
                    menuMap.put(item, menu[i].items[j]);
                    m.add(item);
                }
            }

            rwsNodes = new Vector(500);
            rwsNodeTemplates = new Vector(100);

            toolbar = new JPanel(new FlowLayout());
            toolbar.setPreferredSize(new Dimension(WIDTH, 40));
            toolbar.setBackground(Color.white);
            toolbar.setBorder(BorderFactory.createLineBorder(Color.black));

            templatebar = new JPanel(new FlowLayout());
            templatebar.setPreferredSize(new Dimension(100, HEIGHT));
            templatebar.setBackground(Color.white);
            templatebar.setBorder(BorderFactory.createLineBorder(Color.black));
            JLabel templatelabel = new JLabel("Components:");
            templatebar.add(templatelabel);

            JLabel connectorsLabel, nodelengthLabel, connectortypeLabel;
            JComboBox connectorsCombo, nodelengthCombo;
            JComboBox connectortypeCombo, actionCombo;

            String [] actions = { "Draw",
                                  "Create␣rules"
            };

            /* The set components */
            String [] basicComponents = {
                "End_section",
                "Track_section",
                "Turnout",
                "Rigid_crossing",
                "Double_slip",
                "Scissors",
```

```
            "Single_R",
            "Single_L"
        };

        /* The size of components */
        String [] nodelengths = {
            "10",
            "20",
            "30",
            "40",
            "50",
            "60",
            "70",
            "80",
            "90",
            "100"
        };

        /* The set types */
        String [] connectortypes = {
            "1",
            "2",
            "3",
            "4",
            "5",
            "6",
            "7",
            "8",
            "9",
            "10",
            "11",
            "12",
            "13",
            "14",
            "15"
        };

        /**
         * GUI stuff
         */
        actionCombo = new JComboBox (actions);
        actionCombo.setSelectedIndex (0);
        actionCombo.addActionListener (this);
        actionCombo.setActionCommand ("action");

        connectorsCombo = new JComboBox (basicComponents);
        connectorsCombo.setSelectedIndex (1);
        connectorsCombo.addActionListener (this);
        connectorsCombo.setActionCommand ("comp");
        connectorsCombo.setEditable (true);
        size = connectorsCombo.getPreferredSize ();
        size.width = 130;
        connectorsCombo.setPreferredSize (size);

```

```
        nodelengthCombo = new JComboBox (nodelengths);
        nodelengthCombo.setSelectedIndex (2);
        nodelengthCombo.addActionListener (this);
        nodelengthCombo.setActionCommand ("nodelength");
        nodelengthCombo.setEditable (true);
        size = nodelengthCombo.getPreferredSize ();
        size.width = 50;
        nodelengthCombo.setPreferredSize (size);

        connectortypeCombo = new JComboBox (connectortypes);
        connectortypeCombo.setSelectedIndex (0);
        connectortypeCombo.addActionListener (this);
        connectortypeCombo.setActionCommand ("connectortype");
        connectortypeCombo.setEditable (true);
        size = connectortypeCombo.getPreferredSize ();
        size.width = 50;
        connectortypeCombo.setPreferredSize (size);

        JButton newTemplateButton = new JButton ("Create␣new␣component");
        newTemplateButton.addActionListener (this);
        newTemplateButton.setActionCommand ("newTemplate");

        JLabel absoluteLengthLabel = new JLabel ("Abs.␣length:␣");
        JCheckBox absoluteLengthCheckbox = new JCheckBox ();
        absoluteLengthCheckbox.addItemListener (this);
        absoluteLengthCheckbox.setActionCommand ("absoluteLength");

        JButton writeXMLFileButton = new JButton ("Write␣XML");
        writeXMLFileButton.addActionListener (this);
        writeXMLFileButton.setActionCommand ("writeXML");

        connectorsLabel = new JLabel ("Component:␣");
        nodelengthLabel = new JLabel ("Node␣length:␣");
        connectortypeLabel = new JLabel ("Connector␣type:␣");
        toolbar.add (connectorsLabel);
        toolbar.add (connectorsCombo);
        toolbar.add (nodelengthLabel);
        toolbar.add (nodelengthCombo);
        toolbar.add (newTemplateButton);
        toolbar.add (actionCombo);

        toolbar.add (absoluteLengthLabel);
        toolbar.add (absoluteLengthCheckbox);

        toolbar.add (connectortypeLabel);
        toolbar.add (connectortypeCombo);

        frame.getContentPane ().add (toolbar, BorderLayout.NORTH);
        frame.getContentPane ().add (templatebar, BorderLayout.WEST);

        panel = new BackgroundPanel (this);
        panel.addMouseListener (this);
        panel.addMouseMotionListener (this);

```

```
        insets = panel.getInsets ();

        statusbar = new JLabel ("");
        statusbar.setPreferredSize (new Dimension (WIDTH, 20));
        statusbar.setHorizontalAlignment (JLabel.CENTER);
        statusbar.setOpaque (true);

        /* Add JScrollPane */
        scroller = new JScrollPane (panel);
        scroller.setPreferredSize (new Dimension (900, 600));
        scroller.setWheelScrollingEnabled (true);

        frame.setJMenuBar (menuBar);
        frame.getContentPane ().add (statusbar, BorderLayout.SOUTH);
        frame.getContentPane ().add (scroller, BorderLayout.CENTER);

        frame.addKeyListener (this);

        frame.pack ();
        frame.setVisible (true);

        /* Popup menus */
        popup = new JPopupMenu ();
        JMenuItem connectToCPN = new JMenuItem ("Connect␣to␣CPN␣node");
        connectToCPN.setActionCommand ("connectToCPN");
        connectToCPN.addActionListener (this);

        JMenuItem createMultiple = new JMenuItem ("Create␣multiple␣nodes");
        createMultiple.setActionCommand ("createMultiple");
        createMultiple.addActionListener (this);

        JMenuItem connectNode = new JMenuItem ("Connect␣to␣other␣node");
        connectNode.setActionCommand ("connectNode");
        connectNode.addActionListener (this);

        JMenuItem deleteNode = new JMenuItem ("Delete␣node");
        deleteNode.setActionCommand ("deleteNode");
        deleteNode.addActionListener (this);

        popup.add (connectToCPN);
        popup.add (createMultiple);
        popup.add (deleteNode);
        popup.add (connectNode);

        nodePopup = new JPopupMenu ();
        JMenuItem changeHelpText = new JMenuItem ("Change␣help␣text");
        changeHelpText.setActionCommand ("changeHelpText");
        changeHelpText.addActionListener (this);

        nodePopup.add (changeHelpText);
    }

    /**
     * Calculate the largest X and Y values of connectors. The area of
```

```
     * RWSNodes.
     */
    public void calculateLargestXY () {
        RWSNode rwsnode;
        int local_largestX, local_largestY;
        int nrOFComp = panel.getComponentCount ();
        for (int i = 0; i< nrOFComp; i++){
            rwsnode = (RWSNode) panel.getComponent (i);
            local_largestX = rwsnode.calculateLargestX ();
            local_largestY = rwsnode.calculateLargestY ();
            if (local_largestX > global_largestX)
                global_largestX = local_largestX;
            if (local_largestY > global_largestY)
                global_largestY = local_largestY;
        }
    }

   /**
     * Calculate the smallest X and Y values of connectors. The area
     * of RWSNodes.
     */
    public void calculateSmallestXY(){
        RWSNode rwsnode;
        Dimension d = panel.getPreferredSize();
        global_smallestY= d.height*3;
        global_smallestX = d.width*3;
        int local_smallestX, local_smallestY;
        int nrOFComp = panel.getComponentCount();

        for (int i = 0; i< nrOFComp; i++){
            rwsnode = (RWSNode) panel.getComponent(i);
            local_smallestX = rwsnode.calculateSmallestX();
            local_smallestY = rwsnode.calculateSmallestY();
            if (local_smallestX < global_smallestX)
                global_smallestX = local_smallestX;
            if (local_smallestY < global_smallestY)
                global_smallestY = local_smallestY;
        }
    }

    /**
     * Calculate Mean of RWSNodes areas. Use these values to calculate
     * xmlX and xmlY. We want our final Petri net to be in the center
     * of Design/cpn.
     */
    public void calculateMean(){
        meanX = (global_largestX + global_smallestX)/2;
        meanY = (global_largestY + global_smallestY)/2;
    }

    /**
     * Calculate the angle between the current starting point and the
     * current cursor position
     */
```

```
    public int calculateAngle(int ab, int bc){
        double hyp = Math.sqrt(Math.pow((double) ab, 2.0) +
                               Math.pow((double) bc, 2.0));
        return (int) Math.toDegrees(Math.acos(ab / hyp));
    }

    /**
     * Interface method
     */
    public void mouseClicked(MouseEvent e){}

    /**
     * Interface method
     */
    public void mousePressed(MouseEvent e){
        switch (e.getButton()) {
        case MouseEvent.BUTTON1:
            if (!RWSNode.drawing)
                recordStartingPoint(e.getX(), e.getY());
            else {
                if ((e.getModifiers () & InputEvent.CTRL_MASK) > 0)
                    createNewNode(mouseX, mouseY, false, false);
                else
                    createNewNode(e.getX(), e.getY(), false, false);
            }

            break;
        case MouseEvent.BUTTON3:

            if (RWSConnector.selectedConnector != null){
                RWSConnector.selectedConnector.unsetActive();
                RWSConnector.selectedConnector = null;
            }
            if (RWSNode.selectedNode != null){
                RWSNode.selectedNode.setStatus (RWSNode.INACTIVE);
                RWSNode.selectedNode = null;
            }
            RWSNode.drawing = false;
            break;
        }
        panel.repaint ();
    }

    /**
     * Interface method
     */
    public void mouseReleased (MouseEvent e) {}

    /**
     * Interface method
     */
    public void mouseEntered (MouseEvent e) {}

    /**
```

```
     * Interface method
     */
    public void mouseExited (MouseEvent e) {}

    /**
     * Make sure we always know where the cursor is
     * This is used for creating a dotted line to ease
     * the drawing process.
     */
    public void mouseMoved (MouseEvent e) {
        int sX, sY; /* Starting point */
        int eX, eY; /* Calculated (?) ending point */

        if (RWSConnector.selectedConnector != null) {
            sX = RWSConnector.selectedConnector.externalCenterX ();
            sY = RWSConnector.selectedConnector.externalCenterY ();
        }
        else{
            sX = startX;
            sY = startY;
        }

        int ab = Math.abs(e.getX () - sX);
        int bc = Math.abs(e.getY () - sY);

        if ((e.getModifiers () & InputEvent.CTRL_MASK) > 0) {
            /* CTRL is down, calculate the angle in a set number of degrees */
            if (ab < bc) {
                eX = sX;
                eY = e.getY ();
            }
            else {
                eY = sY;
                eX = e.getX ();
            }
        }
        else {
            eX = e.getX ();
            eY = e.getY ();
        }
        statusbar.setText ("Mouse: " + e.getX () + ", " + e.getY () +
                           " - Angle: " + calculateAngle (ab, bc));
        drawLine(eX, eY);
    }

    /**
     * Interface method
     */
    public void mouseDragged (MouseEvent e) { }

    /**
     * Interface method
     */
    public void keyPressed (KeyEvent e) {}
```

```

    /**
     * Interface method
     */
    public void keyReleased (KeyEvent e) {}

    /**
     * Interface method
     */
    public void keyTyped (KeyEvent e) {
        if (e.getKeyCode () == KeyEvent.VK_DELETE) {
            deleteSelectedNode ();
        }
    }

    /**
     * Make sure a dotted line is drawn from the current starting
     * point to the current cursor position
     */
    private void drawLine (int mX, int mY) {
        if (! RWSNode.drawing)
            return;
        justTheLine = true;
        mouseX = mX;
        mouseY = mY;
        panel.repaint ();
    }

    /**
     * Return the smallest of two integers
     */
    private int smallest (int x, int y) {
        return (x < y) ? x : y;
    }

    /**
     * Return the largest of two integers
     */
    private int largest (int x, int y) {
        return (x >= y) ? x : y;
    }

    /**
     * Record a starting point for drawing
     */
    public void recordStartingPoint (int x, int y) {
        this.startX = x;
        this.startY = y;
        RWSNode.drawing = true;

        if (RWSConnector.selectedConnector != null) {
            RWSConnector.selectedConnector.unsetActive ();
            RWSConnector.selectedConnector = null;
        }
```

```
    }

    /**
     * Resize scroller's dimension
     */
    public void rwsEditorResize (int X, int Y) {
        Dimension preSize = new Dimension (X,Y);

        panel.size = preSize;
        panel.setSize (preSize);

        panel.validate ();
        panel.repaint ();
    }

    /**
     * Create a line of multiple RWSNodes
     */
    boolean rwsConnectMultiple (int num) {
        boolean saveState;

        if (RWSConnector.selectedConnector == null ||
            RWSConnector.selectedConnector.node.numberOfConnectors() != 2)
            return false;

        RWSNode node = RWSConnector.selectedConnector.node;
        startX = RWSConnector.selectedConnector.externalCenterX ();
        startY = RWSConnector.selectedConnector.externalCenterY ();
        int index = RWSConnector.selectedConnector.index == 0 ? 1 : 0;
        int endX = node.connectors [index].externalCenterX ();
        int endY = node.connectors [index].externalCenterY ();
        int diffX = startX - endX;
        int diffY = startY - endY;
        saveState = RWSNode.drawing;
        RWSNode.drawing = true;

        for (int i = 0; i < num; i++) {
            createNewNode (startX + diffX, startY + diffY, false, false);
            startX = startX + diffX;
            startY = startY + diffY;
        }

        RWSNode.drawing = saveState;
        return true;
    }

    /**
     * Well... clears the work space
     */
    public void clearWorkspace () {
        panel.removeAll ();
        rwsNodes.clear ();
        panel.repaint ();
    }
```

```

    /**
     * Removes an RWSNode (atomic component)
     */
    public void deleteSelectedNode () {
        if (RWSNode.selectedNode == null)
            return;
        panel.remove (RWSNode.selectedNode);
        rwsNodes.remove (RWSNode.selectedNode);
        for (int i = 0; i < RWSNode.selectedNode.connectors.length; i++) {
            if (RWSNode.selectedNode.connectors [i].neighbour != null)
                RWSNode.selectedNode.connectors [i].
                    neighbour.neighbour = null;
        }
        panel.repaint();
    }

    /**
     * Clears all templates
     */
    public void clearTemplates () {
        rwsNodeTemplates.clear ();
        templatebar.removeAll ();
        templatebar.repaint ();
    }

    /**
     * Wrapper that also sets starting point
     */
    public void createNewNode (int startX, int startY, int endX, int endY,
                               boolean createNodeTemplate,
                               boolean absoluteLength){
        recordStartingPoint(startX, startY);
        createNewNode(endX, endY, createNodeTemplate, absoluteLength);
    }

    /**
     * Create a new RWSNode (atomic component). Either a template, in
     * which case a new node is created from scratch, or a
     * specification node in which case a template node is copied
     */
    public void createNewNode (int x, int y, boolean createNodeTemplate,
                               boolean absoluteLength) {
        RWSNode r;
        if (createNodeTemplate) {
            r = new RWSNode(0 + (RWSConnector.DOTDIAM / 2),
                            RWSNode.RWSNODELENGTH / 2 +
                            (RWSConnector.DOTDIAM / 2),
                            RWSNode.RWSNODELENGTH * 2,
                            RWSNode.RWSNODELENGTH / 2 +
                            (RWSConnector.DOTDIAM / 2),
                            currentNumberOfConnectors, currentComponent,
                            RWSNode.selectedNode);
            rwsNodeTemplates.add (r);
```

```
            JPanel np = new JPanel (null);
            np.setPreferredSize (new Dimension (RWSNode.RWSNODELENGTH +
                                                RWSConnector.DOTDIAM,
                                                RWSNode.RWSNODELENGTH +
                                                RWSConnector.DOTDIAM));
            np.addMouseMotionListener (this);
            np.add (r);
            templatebar.add (np);
            templatebar.validate ();
            RWSNode.selectedNode = null;
        }
        else if (RWSNode.drawing) {
            if (RWSNode.selectedTemplateNode == null) {
                /* We don't have any template, return. */
                RWSNode.drawing = false;
                return;
            }

            if (RWSConnector.selectedConnector != null &&
                RWSConnector.selectedConnector.isConnected ())
                /* The selected connector is not available */
                return;

            r = new RWSNode(RWSNode.selectedTemplateNode);

            r.initNode (this.startX, this.startY, x, y,
                        RWSNode.selectedTemplateNode.numberOfConnectors (),
                        RWSNode.selectedNode, false, absoluteLength ||
                        alwaysAbsolute);
            r.setTemplate (false);
            r.copyConnectorProperties (RWSNode.selectedTemplateNode);
            r.addConnectors ();
            rwsNodes.add (r);
            panel.add (r);
        }
    }

    /**
     * Debug method. Prints out information about this node's connectors
     */
    void testConnectors (RWSConnector [] connectors) {
        if (connectors != null && connectors [0] != null)
            for (int i = 0; i < connectors.length; i++)
                System.err.println("connectors:␣" +
                                   connectors[i].index + ",␣" +
                                   connectors[i].connectorType + ",␣(" +
                                   connectors[i].externalCenterX() + ",␣" +
                                   connectors[i].externalCenterY() + ")");
        else
            System.err.println("None␣yet...");
    }

    /**
     * Interface method. Handles most actions
```

```
     */
    public void actionPerformed (ActionEvent e) {
        /* Add railway domain information */
        if(e.getSource() instanceof JComboBox){
            JComboBox box = (JComboBox) e.getSource();
            try{
                if(box.getActionCommand() == "comp"){

                    if(box.getSelectedItem().equals("Track_section")){
                       currentComponent = "Track_Section";
                       currentNumberOfConnectors = 2;
                    }
                    else if(box.getSelectedItem().equals("Turnout")){
                        currentComponent = "Turnout";
                        currentNumberOfConnectors = 3;
                    }
                    else if(box.getSelectedItem().equals("Rigid_crossing")){
                        currentComponent = "Rigid_crossing";
                        currentNumberOfConnectors = 4;
                    }
                    else if(box.getSelectedItem().equals("Double_slip")){
                        currentComponent = "Double_slip";
                        currentNumberOfConnectors = 4;
                    }
                    else if( box.getSelectedItem().equals("End_section")){
                        currentComponent = "End_section";
                        currentNumberOfConnectors = 1;
                    }
                    /* Crossovers, different turnout arrangements */
                    else if( box.getSelectedItem().equals("Scissors")){
                        currentComponent = "Scissors";
                        currentNumberOfConnectors = 4;
                    }
                    else if( box.getSelectedItem().equals("Single_R")){
                        currentComponent = "Single_R";
                        currentNumberOfConnectors = 4;
                    }
                    else if( box.getSelectedItem().equals("Single_L")){
                        currentComponent = "Single_L";
                        currentNumberOfConnectors = 4;
                    }
                }

                else if (box.getActionCommand () == "nodelength")
                    RWSNode.setDefaultNodeLength (
                        Integer.parseInt((String) box.getSelectedItem()));
                else if (box.getActionCommand () == "connectortype")
                    RWSConnector.selectedTemplateConnector.setConnectorType (
                        Integer.parseInt((String) box.getSelectedItem()));
                else if (box.getActionCommand () == "action")
                    action = box.getSelectedIndex();
            }
            catch(Exception ex){
                ex.printStackTrace();
```

```
            }
        }

        else if(e.getSource() instanceof JButton){
            JButton button = (JButton) e.getSource();
            try{
                if(button.getActionCommand() == "newTemplate")
                    createNewNode(0, 0, true, false);
            }
            catch(Exception ex){
                ex.printStackTrace();
            }
        }
        else if(e.getSource() instanceof JMenuItem){
            JMenuItem source = ( JMenuItem ) e.getSource ();
            MenuItem item = ( MenuItem ) menuMap.get ( source );

            if (source.getActionCommand () != null &&
                source.getActionCommand () == "connectToCPN") {
                /**
                 * A requested is made to connect the interface (connector)
                 * to it's CPN counterpart
                 */
                ConnectConnectorToCPNNodeFrame cc =
                    new ConnectConnectorToCPNNodeFrame (
                        RWSConnector.selectedTemplateConnector);
                cc.pack ();
                cc.setVisible (true);
            }
            else if (source.getActionCommand () != null &&
                     source.getActionCommand () == "createMultiple") {
                CreateMultipleNodesFrame cm =
                    new CreateMultipleNodesFrame (this);
            }
            else if ( source.getActionCommand () != null &&
                      source.getActionCommand () == "deleteNode" ) {
                deleteSelectedNode();
            }
            else if ( source.getActionCommand () != null &&
                      source.getActionCommand () == "changeHelpText" ) {
                new ChangeToolTipText (RWSNode.tmpSelected);
            }
            else if ( source.getActionCommand () != null &&
                      source.getActionCommand () == "connectNode" ) {
                RWSConnector.connectNext = true;
            }
            else {
                String debug;

                switch(item.getKey()){
                case Menu.FILE_NEW:
                    debug = "File -> New";
                    clearWorkspace();
                    break;
```

```
                case Menu.FILE_OPEN_ALL:
                    debug = "File␣->␣Open␣Project";
                    chooser = new JFileChooser();
                    chooser.setDialogTitle("Open␣Project");
                    chooser.setFileFilter(new RWSFileFilter());
                    if (chooser.showOpenDialog(panel) ==
                        JFileChooser.APPROVE_OPTION) {
                        clearWorkspace ();
                        clearTemplates ();
                        clearWorkspace ();
                        RWSEditor.xmlUtils.openProject (
                            chooser.getSelectedFile ().getPath (),
                            templatebar, panel, this);
                    }
                    break;
                case Menu.FILE_SAVE_ALL:
                    debug = "File␣->␣Save␣Project";
                    chooser = new JFileChooser();
                    chooser.setDialogTitle("Save␣Project");
                    chooser.setFileFilter(new RWSFileFilter());
                    if (chooser.showSaveDialog (panel) ==
                        JFileChooser.APPROVE_OPTION) {
                        RWSEditor.xmlUtils.saveProject (
                            chooser.getSelectedFile ().getPath (),
                            panel, templatebar);
                    }
                    break;
                case Menu.FILE_OPEN_CPN:
                    debug = "File␣->␣Open␣CPN␣Component";
                    chooser = new JFileChooser();
                    chooser.setDialogTitle("Open␣CPN␣component");
                    chooser.setFileFilter(new XMLFileFilter());
                    if (chooser.showOpenDialog(panel) ==
                        JFileChooser.APPROVE_OPTION) {
                        RWSEditor.xmlUtils.readXML (
                            chooser.getSelectedFile().getPath(),
                            chooser.getSelectedFile().getName());
                    }
                    break;
                case Menu.FILE_SAVE_CPN:
                    debug = "File␣->␣Save␣CPNet";
                    chooser = new JFileChooser();
                    chooser.setDialogTitle("Save␣CPN␣component");
                    chooser.setFileFilter(new XMLFileFilter());
                    if (chooser.showSaveDialog(panel) ==
                        JFileChooser.APPROVE_OPTION) {
                        calculateLargestXY ();
                        calculateSmallestXY ();
                        calculateMean ();
                        RWSEditor.xmlUtils.initPrintXml (
                            (RWSNode) panel.getComponent (0),
                            chooser.getSelectedFile ().getPath ());
                    }
                    break;
```

```
                case Menu.FILE_EXIT:
                    System.exit(0);
                    debug = "File␣->␣Quit";
                    break;
                case Menu.EDIT_UNDO:
                    debug = "Edit␣->␣Undo";
                    break;
                case Menu.EDIT_DELETE:
                    debug = "Edit␣->␣Delete";
                    deleteSelectedNode();
                    break;
                case Menu.EDIT_REDO:
                    debug = "Edit␣->␣Redo";
                    break;
                case Menu.EDIT_CLEAR:
                    debug = "Edit␣->␣Clear";
                    clearWorkspace();
                    break;
                case Menu.EDIT_CLEAR_TEMPLATES:
                    debug = "Edit␣->␣Clear␣templates";
                    clearTemplates();
                    break;
                case Menu.TOOLS_OPTIONS:
                    debug = "Tools␣->␣Options";
                    break;
                case Menu.TOOLS_RESIZE:
                    debug = "Tools␣->␣Resize";
                    Resize pix = new Resize ( this );
                    break;

                default:
                    debug = "Switch␣->␣Default";
                }
                if (RWSEditor.DEBUG) System.err.println(debug);
            }
        }
    }

    /**
     * Interface method
     */
    public void itemStateChanged (ItemEvent e) {
        if (e.getSource() instanceof JCheckBox) {
            JCheckBox box = (JCheckBox) e.getSource ();
            if (box.getActionCommand () == "absoluteLength")
                alwaysAbsolute = e.getStateChange () == ItemEvent.SELECTED;
        }
    }

    /**
     * Returns an array of Menu objects
     */
    private Menu [] getMenu () {
        Menu [] menu = new Menu [3];
```

```
        menu [0] = new Menu("File", 10);
        menu [0].addItem ("New", Menu.FILE_NEW, KeyEvent.VK_N,
                          ActionEvent.CTRL_MASK);
        menu [0].addItem ("Open␣Project", Menu.FILE_OPEN_ALL, KeyEvent.VK_O,
                          ActionEvent.CTRL_MASK);
        menu [0].addItem ("Save␣Project", Menu.FILE_SAVE_ALL, KeyEvent.VK_S,
                          ActionEvent.CTRL_MASK);
        menu [0].addItem ("Open␣CPN␣Component", Menu.FILE_OPEN_CPN);
        menu [0].addItem ("Save␣CPNet", Menu.FILE_SAVE_CPN);
        menu [0].addItem ("Quit", Menu.FILE_EXIT, KeyEvent.VK_Q,
                          ActionEvent.CTRL_MASK);

        menu [1] = new Menu ("Edit", 5);
        menu [1].addItem ("Undo", Menu.EDIT_UNDO);
        menu [1].addItem ("Redo", Menu.EDIT_REDO);
        menu [1].addItem ("Delete", Menu.EDIT_DELETE);
        menu [1].addItem ("Clear␣workspace", Menu.EDIT_CLEAR);
        menu [1].addItem ("Clear␣all␣templates", Menu.EDIT_CLEAR_TEMPLATES);

        menu [2] = new Menu ("Tools", 2);
        menu [2].addItem ("Options", Menu.TOOLS_OPTIONS);
        menu [2].addItem ("Resize", Menu.TOOLS_RESIZE);

        return menu;
    }

}

/**
 * Class for easing the menu handling. The menu in this case being the
 * standard menu line docked in the topmost section of GUI programs.
 */
class Menu {

    static final int FILE_NEW      =   0;
    static final int FILE_OPEN     =   1;
    static final int FILE_OPEN_CPN =   2;
    static final int FILE_SAVE     =   4;
    static final int FILE_SAVE_CPN =   5;
    static final int FILE_EXIT     =   7;
    static final int FILE_OPEN_ALL =   8;
    static final int FILE_SAVE_ALL =   9;
    static final int EDIT_UNDO     = 100;
    static final int EDIT_REDO     = 101;
    static final int EDIT_DELETE   = 102;
    static final int EDIT_CLEAR    = 103;
    static final int EDIT_CLEAR_TEMPLATES = 104;
    static final int TOOLS_OPTIONS = 200;
    static final int TOOLS_RESIZE  = 201;

    private String text;
    MenuItem [] items;

    Menu (String text, int length) {
```

```
        this.text = text;
        items = new MenuItem [length];
    }

    void addItem (String text, int key) {
        try{
            items [getTopMostItem ()] = new MenuItem (text, key);
        }
        catch (ArrayIndexOutOfBoundsException e) {
            System.err.println ("Not␣enough␣room␣for␣element␣'" + text +
                                "'␣in␣menu␣'" + this.text + "'.");
        }
    }

    void addItem (String text, int key, int sc, int modifier) {
        try {
            items [getTopMostItem ()] = new MenuItem (text, key, sc, modifier);
        }
        catch (ArrayIndexOutOfBoundsException e) {
            System.err.println ("Not␣enough␣room␣for␣element␣'" + text +
                                "'␣in␣menu␣'" + this.text + "'.");
        }
    }

    int getTopMostItem () {
        for (int i = 0; i < items.length; i++)
            if (items [i] == null)
                return i;
        return -1;
    }

    String getText () {
        return this.text;
    }
}

/**
 * An item in the menu
 */
class MenuItem {

    private String text;
    private int key;
    private int sc;
    private int modifier;

    MenuItem (String text, int key) {
        this.text = text;
        this.key = key;
    }

    MenuItem (String text, int key, int sc, int modifier) {
        this.text = text;
        this.key = key;
```

```
        this.sc = sc;
        this.modifier = modifier;
    }

    int getKey () {
        return this.key;
    }

    String getText () {
        return this.text;
    }

    int getSC () {
        return sc;
    }

    int getModifier () {
        return modifier;
    }

    boolean hasSC () {
        return sc != 0;
    }

    boolean hasModifier () {
        return modifier != 0;
    }
}
```

Listing 3: RWSNode.java

```
import java.io.Serializable;
import java.awt.Panel;
import java.awt.event.*;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.BasicStroke;
import java.awt.Rectangle;
import java.awt.Point;

import javax.swing.JPanel;
import org.w3c.dom.*;
import org.xml.sax.*;

/**
 * RWSNode objects are the atomic components of the specification
 * language. They have a set of RWSConnectors, which are the interface
 * nodes of the specification language.
 */
public class RWSNode extends JPanel implements MouseListener,
                                               MouseMotionListener,
                                               Serializable {
```

```

    /* Array of connectors */
    protected RWSConnector [] connectors;

    /* id # and counter */
    protected int id;
    private static int counter = 0;

    /* For moving the center point */
    protected boolean moving = false;

    protected boolean mark = false;

    /* The type of Node (railway domain) ie: Road, Switch etc. */
    protected String componentType;
    protected String cpnComponentType;

    /* Dimensions */
    static protected int RWSNODELENGTH = 30;
    static private int RWSNODEWIDTH = 4;
    static private int ENDPOINTDIAM = RWSConnector.DOTDIAM;

    /* Colors */
    final static Color USED_ENDPOINTCOLOR = Color.black;
    final static Color UNUSED_ENDPOINTCOLOR = Color.red;
    final static Color ACTIVE_RWSNODECOLOR = Color.blue;
    final static Color RWSNODECOLOR = Color.black;
    final static Color NODE_FILLCOLOR = new Color(0x80, 0x80, 0x80);
    final static Color NODE_INST_COLOR = Color.lightGray;

    /* Status */
    final public static int INACTIVE = 0;
    final public static int ACTIVE = 1;
    private int status;

    public static RWSNode selectedNode, selectedTemplateNode, tmpSelected;
    public static boolean drawing = false;

    /* Mouse buttons */
    final protected static int MOUSE_LEFT = 0;
    final protected static int MOUSE_MIDDLE = 1;
    final protected static int MOUSE_RIGHT = 2;

    /**
     * positionMark is used for determining whether this node has
     * been given a position while printing the CPN XML.
     * Used in conjunction with XMLUtils.currentMark for reusability.
     * xmlX, xmlY is the "origo" for this XML component.
     * relX, relY are the coordinates from the CPN component acting as "origo".
     */
    protected boolean positionMark = false;
    protected int xmlX, xmlY, relX, relY;

    /* Coordinates of the center of this node */
```

```
    private int centerX, centerY;

    /* If only one connector, we need an extra line */
    private int endP1X, endP2X, endP1Y, endP2Y;

    /**
     * If this node is not a template node,
     * this is a reference to it's template
     */
    protected RWSNode template;

    /**
     * If this node is a template, it should have other
     * properties...
     */
    protected boolean isTemplate = true;

    /* Where is this node located? */
    private Rectangle externalCoordinates = new Rectangle();

    private int[] connectorX;
    private int[] connectorY;

    /**
     * This is used when the node is a template, in which
     * case RWSNODELENGTH is set after this.
     */
    protected int nodeLength;


    /**
     * Constructor 1.
     */
    public RWSNode () {
        setOpaque (false);
        setLayout (null);
        addMouseListener (this);
    }

    /**
     * Constructor 2. Might be outdated.
     */
    RWSNode(int i, int nr){
        connectors = new RWSConnector [i];
        RWSConnector r;

        for(int j=0;j<connectors.length;j++){
            r = new RWSConnector();
            connectors [j] = r;
            r.node = this;
        }
        id = nr;
    }

```

```
    /**
     * Constructor 3. When we copy properties from a template.
     */
    RWSNode(RWSNode template){
        id = counter;
        counter++;

        this.template = template;
        int edges = template.numberOfConnectors();
        setOpaque (false);
        setLayout (null);
        componentType = template.componentType;
        setToolTipText (template.getToolTipText ());
        connectorX = new int [edges];
        connectorY = new int [edges];

        connectors = new RWSConnector [edges];

        addMouseListener(this);
    }

    /**
     * Constructor 4. When we create a template.
     */
    RWSNode(int startX, int startY, int endX, int endY, int edges,
            String name, RWSNode prev){
        id = counter;
        counter++;

        setOpaque (false);
        setLayout (null);
        componentType = name;
        setToolTipText (componentType);
        connectorX = new int [edges];
        connectorY = new int [edges];

        addMouseListener (this);

        connectors = new RWSConnector [edges];

        initNode (startX, startY, endX, endY, edges, prev, true, false);

        addConnectors ();

        selectedNode = null;
        RWSConnector.selectedConnector = null;
    }


    /**
     * Adds all containers to this node's panel.
     */
    protected void addConnectors (){
        for(int i = 0; i < connectors.length; i++)
```

```
            add (connectors [i], -1);
    }

    /**
     * Initializes the node...
     */
    protected void initNode (int startX, int startY, int endX, int endY,
                             int edges, RWSNode prev, boolean template,
                             boolean absoluteLength){
        setTemplate (template);
        RWSConnector previousConnector = null;
        if( RWSConnector.selectedConnector != null && ! isTemplate){
            startX = RWSConnector.selectedConnector.externalCenterX();
            startY = RWSConnector.selectedConnector.externalCenterY();
            previousConnector = RWSConnector.selectedConnector;
        }
        calculatePositions(startX, startY, endX, endY, edges, absoluteLength);
        this.setBounds(externalCoordinates);

        if (! isTemplate){
            selectedNode = this;
            if (previousConnector != null)
                connectConnectors (
                    connectors [RWSConnector.
                                selectedTemplateConnectorIndex ()],
                    previousConnector );
        }
        else
            nodeLength = RWSNODELENGTH;
    }

    /**
     * Set whether this node is a template or not.
     */
    protected void setTemplate(boolean set){
        for(int i=0;i<connectors.length;i++)
            if ( connectors [i] != null)
                connectors [i].isTemplate = set;
        isTemplate = set;
    }

    /**
     * Workhorse method. Here all the coordinates of the connectors
     * are calculated. This is done by first calculating the coordinates
     * of the point located RWSNODELENGTH / 2 from (startX, startY) on the
     * line towards (endX, endY). This is point e in the figure:
     *
     *           c
     *          /|
     *      e /  |
     *      /    |
     *    / |    |
     *  /___|____|
     * a    d    b
```

```
     *
     * Thereafter, according to how many connectors this node has, an
     * equal number of points evenly placed in a circle around e with a
     * radius of ae, starting with the starting point.
     */
    private void calculatePositions(int startX, int startY, int endX, int endY,
                                    int num, boolean absoluteLength){
        int ab = Math.abs(endX - startX);
        int bc = Math.abs(endY - startY);
        double hyp = Math.sqrt(Math.pow((double) ab, 2.0) +
                               Math.pow((double) bc, 2.0));
        double angle = Math.acos(ab / hyp);
        double radius;
        int ad, de;
        if ( ! absoluteLength )
            radius = RWSNODELENGTH / 2;
        else
            radius = hyp / 2;
        ad = (int) Math.round(radius * Math.cos(angle));
        de = (int) Math.round(radius * Math.sin(angle));

        /* external coordinates for center of node */
        int extCX = (startX < endX) ? startX + ad : startX - ad; // center x
        int extCY = (startY < endY) ? startY + de : startY - de; // center y
        int adjustment = (startY < endY) ? -1 : 1;

        if (RWSEditor.DEBUG)
            System.err.println("(" + extCX + ",␣" + extCY + ")");

        /* make cX, cY origo */
        int x = startX - extCX;
        int y = startY - extCY;
        double calcX, calcY;
        if ( Math.abs(x) > Math.abs(radius) )
            calcX = ( x < 0 ) ? - 1.0 : 1.0;
        else
            calcX = x / radius;

        /* calculate angle between ((0,0),(5,0)) and ((0,0),(x,y)) */
        angle = Math.acos(calcX);

        /* How many degrees between each point? */
        double degrees = (2 * Math.PI) / num;

        int highestX, highestY, lowestX, lowestY;
        if(num == 1){
            /**
             * If there is only one connector, we
             * still need this node to occupy space.
             */
            if(extCX < startX){
                lowestX = extCX;
                highestX = startX;
            }
```

```
            else{
                lowestX = startX;
                highestX = extCX;
            }
            if(extCY < startY){
                lowestY = extCY;
                highestY = startY;
            }
            else{
                lowestY = startY;
                highestY = extCY;
            }
        }
        else{
            lowestX = highestX = startX;
            lowestY = highestY = startY;
        }

        connectorX [0] = startX;
        connectorY [0] = startY;

        if(num > 1){
            for(int i=1;i<num;i++){

                angle += degrees;

                connectorX [i] = (int) (radius * Math.cos(angle) + extCX);
                connectorY [i] = (int) (adjustment * (radius * Math.sin(angle))
                                        + extCY);

                if(connectorX [i] < lowestX)
                    lowestX = connectorX [i];
                else if(connectorX [i] > highestX)
                    highestX = connectorX [i];
                if(connectorY [i] < lowestY)
                    lowestY = connectorY [i];
                else if(connectorY [i] > highestY)
                    highestY = connectorY [i];

            }
        }
        else{
            /* We have only one connector */
            angle += Math.PI / 2;
            endP1X = (int) ((ENDPOINTDIAM / 2) * Math.cos(angle) + extCX) -
                lowestX + borderWidth();
            endP1Y = (int) (adjustment * ((ENDPOINTDIAM / 2) * Math.sin(angle))
                            + extCY) - lowestY + borderWidth();

            angle += Math.PI;
            endP2X = (int) ((ENDPOINTDIAM / 2) * Math.cos(angle) + extCX) -
                lowestX + borderWidth();
            endP2Y = (int) (adjustment * ((ENDPOINTDIAM / 2) * Math.sin(angle))
                            + extCY) - lowestY + borderWidth();
```

```
        }

        /**
         * Since the "starting connector" can be any one connector
         * from connectors [0] to connectors [ connectors.length - 1 ],
         * this has to be tweaked.
         */
        int j = RWSConnector.selectedTemplateConnectorIndex();
        for(int i=0;i<num;i++){
            if(isTemplate)
                connectors [i] = new RWSConnector (connectorX [i] - lowestX +
                                                   borderWidth(),
                                                   connectorY [i] - lowestY +
                                                   borderWidth(),
                                                   this, i, true);
            else {
                connectors [j] = new RWSConnector (connectorX [i] - lowestX +
                                                   borderWidth(),
                                                   connectorY [i] - lowestY +
                                                   borderWidth(),
                                                   this, j, false);
            }
            if(adjustment < 0){
                j--;
                if(j < 0)
                    j = num - 1;
            }
            else{
                j++;
                if(j >= num)
                    j = 0;
            }
        }

        /* Clean up */
        for (int i = 0; i < num; i++) {
            connectorX [i] = connectors [i].getP ().x +
                lowestX - borderWidth ();
            connectorY [i] = connectors [i].getP ().y +
                lowestY - borderWidth ();
        }

        /* Set the center point of this node. */
        centerX = extCX - lowestX + borderWidth();
        centerY = extCY - lowestY + borderWidth();

        /* Set the external coordinates */
        externalCoordinates.x = lowestX - borderWidth();
        externalCoordinates.y = lowestY - borderWidth();
        externalCoordinates.width = highestX - lowestX + (borderWidth() * 2);
        externalCoordinates.height = highestY - lowestY + (borderWidth() * 2);
    }

    /**
```

```
     * Rescale this node after a connector has been dragged
     * @param  int  xMove  How much to move the connector horizontally
     * @param  int  yMove  How much to move the connector vertically
     * @param  int  index  The index of the connector in connectors []
     * @since  1.22
     * @return void
     */
    protected void rescaleNode (int xMove, int yMove, int index) {
        int lowestX, lowestY, highestX, highestY;
        int oldX, oldY, newX, newY, xDiff, yDiff;
        int border;

        /**
         * Now, we know that the connector with index index is moved x spaces
         * horizontally and y spaces vertically.
         * Furthermore, we know that connectors [index].internalCenterX ()
         * and connectors [index].internalCenterY () provides the current
         * position of the connector.
         */
        if (index >= 0) {
            oldX = connectors [index].internalCenterX ();
            oldY = connectors [index].internalCenterY ();
        }
        else {
            oldX = centerX;
            oldY = centerY;
        }

        if (xMove > 0 || (oldX + xMove) > 0)
            /* oldX < newX or |xMove| <= oldX */
            newX = oldX + xMove;
        else
            /* |xMove| >= oldX */
            newX = borderWidth ();

        /* And the vertical direction */
        if (yMove > 0 || (oldY + yMove) > 0)
            newY = oldY + yMove;
        else
            newY = borderWidth ();

        /**
         * Find the highest and lowest (x, y) *apart*
         * from connectors [index]
         */
        if (index >= 0) {
            lowestX = centerX + externalCoordinates.x;
            lowestY = centerY + externalCoordinates.y;
            highestX = centerX + externalCoordinates.x;
            highestY = centerY + externalCoordinates.y;
        }
        else {
            lowestX = connectorX [0];
            lowestY = connectorY [0];
```

```
            highestX = connectorX [0];
            highestY = connectorY [0];
        }

        for (int i = 0; i < connectors.length; i++) {
            if (i != index) {
                if (connectorX [i] < lowestX)
                    lowestX = connectorX [i];
                else if (connectorX [i] > highestX)
                    highestX = connectorX [i];
                if (connectorY [i] < lowestY)
                    lowestY = connectorY [i];
                else if (connectorY [i] > highestY)
                    highestY = connectorY [i];
            }
        }

        /**
         * We now have the lowest possible x and y in lowestX and lowestY
         * if we disregard connectors [index]
         * Now, calculate the difference between the points and the difference
         * between the lowest point apart from this to the lowest point of the
         * node.
         */

        int abX = Math.abs (xMove);
        int cdX = lowestX - (externalCoordinates.x + borderWidth ());
        int abY = Math.abs (yMove);
        int cdY = lowestY - (externalCoordinates.y + borderWidth ());

        if (cdX > 0) {
            /* connectors [index] is the sole leftmost point */
            if (xMove > 0) {
                if (abX < cdX) {
                    xDiff = -abX;
                    lowestX -= (cdX - abX);
                }
                else {
                    xDiff = -cdX;
                    if (xMove > externalCoordinates.width)
                        highestX += xMove - (externalCoordinates.width -
                                             (borderWidth () * 2));
                }
            }
            else {
                xDiff = abX;
                lowestX -= (abX + cdX);
            }
        }
        else if (xMove < 0 && abX > (oldX - borderWidth ())) {
            xDiff = abX - oldX + borderWidth ();
            lowestX -= xDiff;
        }
        else { /* externalCoordinates.x remains the same */
```

```
            xDiff = 0;
            if ((externalCoordinates.x + newX) > highestX)
                highestX = externalCoordinates.x + newX;
        }

        /* Repeat for y (should put this in a method) */
        if (cdY > 0) {
            /* connectors [index] is the sole topmost point */
            if (yMove > 0) {
                if (abY < cdY) {
                    yDiff = -abY;
                    lowestY -= (cdY - abY);
                }
                else {
                    yDiff = -cdY;
                    if (yMove > externalCoordinates.height)
                        highestY += yMove - (externalCoordinates.height -
                                             (borderWidth () * 2));
                }
            }
            else {
                yDiff = abY;
                lowestY -= (abY + cdY);
            }
        }
        else if (yMove < 0 && abY > (oldY - borderWidth ())) {
            yDiff = abY - oldY + borderWidth ();
            lowestY -= yDiff;
        }
        else { /* externalCoordinates.y remains the same */
            yDiff = 0;
            if ((externalCoordinates.y + newY) > highestY)
                highestY = externalCoordinates.y + newY;
        }

        if (index >= 0) {
            connectorX [index] += xMove;
            connectorY [index] += yMove;
        }
        else {
            centerX += xMove;
            if (centerX < borderWidth ())
                centerX = borderWidth ();
            centerY += yMove;
            if (centerY < borderWidth ())
                centerY = borderWidth ();
        }

        for (int i = 0; i < connectors.length; i++)
            if (i != index && xDiff != 0 || yDiff != 0)
                    connectors [i].moveRelative (xDiff, yDiff);

        if (index >= 0) {
            centerX += xDiff;
```

```
            centerY += yDiff;
        }

        /* Set the external coordinates */
        externalCoordinates.x = lowestX - borderWidth();
        externalCoordinates.y = lowestY - borderWidth();
        externalCoordinates.width = highestX - lowestX + (borderWidth() * 2);
        externalCoordinates.height = highestY - lowestY + (borderWidth() * 2);

        if (index >= 0) {
            connectors [index].setP (new Point (connectorX [index] -
                                                externalCoordinates.x,
                                                connectorY [index] -
                                                externalCoordinates.y));
        }

        setSize (new Dimension (externalCoordinates.width,
                                externalCoordinates.height));
        setBounds (externalCoordinates.x, externalCoordinates.y,
                   externalCoordinates.width, externalCoordinates.height);

        validate ();
        if (index >= 0)
            connectors [index].validate ();
        repaint ();
    }

    /**
     * Calculate the largest X value of this rwsNode's connectors
     */
    protected int calculateLargestX(){
        int local_largestX = connectorX[0]*3;
        for(int i = 0; i<connectorX.length ; i++){
            if (connectorX[i]*3 > local_largestX)
                local_largestX = connectorX[i]*3;
        }
        return local_largestX;
    }

    /**
     * Calculate the largest Y value of this rwsNode's connectors
     */
    protected int calculateLargestY(){
        Dimension bgSize = RWSEditor.frame.panel.getPreferredSize ();
        int local_largestY = (bgSize.height-connectorY[0])*3;
        for(int i = 0; i<connectorY.length ; i++){
            if ((bgSize.height-connectorY[i])*3 > local_largestY)
                local_largestY = (bgSize.height-connectorY[i])*3;
        }
        return local_largestY;
    }

    /**
     * Calculate the smallest X value of this rwsNode's connectors
```

```
     */
    protected int calculateSmallestX(){
        int local_smallestX = connectorX[0]*3;
        for(int i = 0; i<connectorX.length ; i++){
            if (connectorX[i]*3 < local_smallestX)
                local_smallestX = connectorX[i]*3;
        }
        return local_smallestX;
    }

    /**
     * Calculate the smallest Y value of this rwsNode's connectors
     */
    protected int calculateSmallestY(){
        Dimension bgSize = RWSEditor.frame.panel.getPreferredSize ();
        int local_smallestY = (bgSize.height-connectorY[0])*3;
        for(int i = 0; i<connectorY.length ; i++){
            if ((bgSize.height-connectorY[i])*3 < local_smallestY)
                local_smallestY = (bgSize.height-connectorY[i])*3;
        }
        return local_smallestY;
    }

    /**
     * Copies the properties from the template node.
     * Also (maybe not the Correct[tm] method to do it in..) sets the status
     * of the connectors.
     * Finally (should also probably be done in another method) unset the
     * selected connector if it can't connect to the selected template
     * connector.
     */
    protected void copyConnectorProperties ( RWSNode templateNode ){
        for (int i=0;i<connectors.length;i++) {
            connectors [i].connectorType =
                templateNode.connectors [i].connectorType;
            connectors [i].cpnInterface =
                templateNode.connectors [i].cpnInterface;
        }
        if ( RWSConnector.selectedConnector != null ) {

            if ( ! RWSConnector.selectedConnector.canConnectTo
                 ( RWSConnector.selectedTemplateConnector )||
                 RWSConnector.selectedConnector.isConnected() ){
                RWSConnector.selectedConnector.unsetActive();
                RWSConnector.selectedConnector = null;
                selectedNode = null;
                setSelectedConnector();
            }
        }
        else {
            setSelectedConnector();
        }
    }

```

```
    void setSelectedConnector(){
        drawing = false;

        for (int i=connectors.length-1;i>=0;i--) {
            if (! connectors [i].isConnected() && // This connactor is free
                /* It is not the same as the template connector */
                i != RWSConnector.selectedTemplateConnector.index &&
                /* This connector can connect to the template */
                connectors [i].canConnectTo (
                    RWSConnector.selectedTemplateConnector)) {
                selectedNode = this;
                drawing = true;

                RWSConnector.selectedConnector = connectors [i];
                RWSConnector.selectedConnector.setStatus(RWSConnector.ACTIVE);
                return;
            }
        }
    }

    /**
     * Update this node's status
     */
    public void setStatus(int status){
        this.status = status;
    }

    /**
     * Which mouse button was pressed?
     */
    private int getMouseButton(MouseEvent e){
        switch(e.getButton()){
        case MouseEvent.BUTTON1:
            return MOUSE_LEFT;
        case MouseEvent.BUTTON2:
            return MOUSE_MIDDLE;
        case MouseEvent.BUTTON3:
            return MOUSE_RIGHT;
        }
        return -1;
    }

    /**
     * What happens when this node is clicked.
     * This could also be placed in mousePressed(), have to look at it..
     */
    public void mouseClicked(MouseEvent e){
        RWSNode previous;

        if (getMouseButton (e) == MOUSE_RIGHT) {
            tmpSelected = this;
            RWSEditorFrame.nodePopup.show (RWSEditorFrame.bg,
                                           e.getX () + externalCoordinates.x,
                                           e.getY () + externalCoordinates.y);
```

```
        }

        if (isTemplate) {
            previous = selectedTemplateNode;
            selectedTemplateNode = this;
            RWSNode.setDefaultNodeLength(nodeLength);
        }
        else {
            previous = selectedNode;
            selectedNode = this;
            setStatus(ACTIVE);
            if ( previous != null )
                previous.setStatus(INACTIVE);
        }
        repaint();
        if ( previous != null )
            previous.repaint();
    }

    public void mousePressed(MouseEvent e){}

    public void mouseReleased(MouseEvent e){
        if (moving) {
            if (RWSConnector.hoveringConnector != null)
                /**
                 * We're over a connector, we don't want to
                 * place the center point here
                 */
                return;
            rescaleNode (e.getX () - centerX,
                         e.getY () - centerY, -1);
            moving = false;
        }
    }

    public void mouseEntered(MouseEvent e){}

    public void mouseExited(MouseEvent e){}

    /**
     * Interface method
     */
    public void mouseDragged (MouseEvent e) {
        moving = true;
    }

    /**
     * Interface method
     */
    public void mouseMoved (MouseEvent e) { }

    public Dimension getPreferredSize() {
        return new Dimension(externalCoordinates.width + ENDPOINTDIAM,
                             externalCoordinates.height + ENDPOINTDIAM);
```

```
    }

    /**
     * Paint this node. We need a Graphics2D object since we want a
     * thicker line.
     */
    public void paintComponent(Graphics g){
        Graphics2D g2 = (Graphics2D) g;
        g2.setStroke(new BasicStroke(RWSNODEWIDTH, BasicStroke.CAP_ROUND,
                                     BasicStroke.JOIN_ROUND));
        switch(status){
        case ACTIVE:
            g2.setColor(ACTIVE_RWSNODECOLOR);
            break;
        default:
            g2.setColor(RWSNODECOLOR);
        }
        if(!componentType.equals ("Single_R") &&
           !componentType.equals ("Single_L")) {
            if(componentType.equals("Track_section")){
                g2.drawLine(centerX, centerY,
                            connectors [0].internalCenterX (),
                            connectors [0].internalCenterY ());
                g2.setColor(Color.gray);
                g2.drawLine(centerX, centerY,
                            connectors [1].internalCenterX (),
                            connectors [1].internalCenterY ());
            }
            else {
                for(int i=0;i<connectors.length;i++){
                    g2.drawLine(centerX, centerY,
                                connectors [i].internalCenterX (),
                                   connectors [i].internalCenterY ());
                }
            }
        }
        if(componentType!=null && componentType.equals("Turnout")){
            Graphics2D g3 = (Graphics2D) g;
            g.setColor(NODE_FILLCOLOR);
            g3.setStroke(new BasicStroke(2, BasicStroke.CAP_ROUND,
                                         BasicStroke.JOIN_ROUND));

            int [] polyX = { connectors [1].internalCenterX(),
                             connectors [2].internalCenterX(), centerX };
            int [] polyY = { connectors [1].internalCenterY(),
                             connectors [2].internalCenterY(), centerY };

            g3.fillPolygon(polyX, polyY, 3);
        }

        if(componentType != null && componentType.equals ("Double_slip")){
            Graphics2D g3 = (Graphics2D) g;
            g3.setStroke(new BasicStroke(2, BasicStroke.CAP_ROUND,
                                         BasicStroke.JOIN_ROUND));
```

```
            g.setColor(NODE_INST_COLOR);
            g3.drawLine(connectors[0].internalCenterX(),
                        connectors[0].internalCenterY(),
                        connectors[1].internalCenterX(),
                        connectors[1].internalCenterY());
            g3.drawLine(connectors[2].internalCenterX(),
                        connectors[2].internalCenterY(),
                        connectors[3].internalCenterX(),
                        connectors[3].internalCenterY());
            int [] polyX = { connectors[1].internalCenterX(),
                             connectors[2].internalCenterX(),
                             centerX };
            int [] polyY = { connectors[1].internalCenterY(),
                             connectors[2].internalCenterY(),
                             centerY };
            int [] poly2X = { connectors[0].internalCenterX(),
                              connectors[3].internalCenterX(),
                              centerX };
            int [] poly2Y = { connectors[0].internalCenterY(),
                              connectors[3].internalCenterY(),
                              centerY };
            g.setColor(NODE_FILLCOLOR);

            g3.fillPolygon(polyX, polyY, 3);
            g3.fillPolygon(poly2X, poly2Y, 3);
        }

        if(componentType!=null && componentType.equals("Scissors")){
            g2.drawLine (connectors[0].internalCenterX(),
                         connectors[0].internalCenterY(),
                         connectors[1].internalCenterX(),
                         connectors[1].internalCenterY());
            g2.drawLine (connectors[2].internalCenterX(),
                         connectors[2].internalCenterY(),
                         connectors[3].internalCenterX(),
                         connectors[3].internalCenterY());
        }

        if (componentType != null &&
            (componentType.equals ("Single_R") ||
             componentType.equals ("Single_L"))){
            if (componentType.equals ("Single_R")){
                for (int i = 0; i < connectors.length; i++) {
                    g2.drawLine (centerX, centerY,
                                connectors [i].internalCenterX (),
                                connectors [i].internalCenterY ());
                    i++;
                }
            }
            else {
                for(int i = 1; i<connectors.length; i++){
                    g2.drawLine(centerX, centerY,
                                connectors [i].internalCenterX(),
                                   connectors [i].internalCenterY());
```

```
                        i++;
                    }
                }

                g2.drawLine(connectors[0].internalCenterX(),
                            connectors[0].internalCenterY(),
                            connectors[1].internalCenterX(),
                            connectors[1].internalCenterY());
                g2.drawLine(connectors[2].internalCenterX(),
                            connectors[2].internalCenterY(),
                            connectors[3].internalCenterX(),
                            connectors[3].internalCenterY());
            }

            if(connectors.length == 1){
                g2.drawLine(endP1X, endP1Y, endP2X, endP2Y);
            }
            g = (Graphics) g2;
            super.paintComponent(g);
        }

        /**
         * Return the corner coordinates of this node.
         */
        public int externalEndX(){
            return externalCoordinates.x + externalCoordinates.width;
        }

        /**
         * Return the corner coordinates of this node.
         */
        public int externalEndY(){
            return externalCoordinates.y + externalCoordinates.height;
        }

        /**
         * Return the corner coordinates of this node.
         */
        public int externalStartX(){
            return externalCoordinates.x;
        }

        /**
         * Return the corner coordinates of this node.
         */
        public int externalStartY(){
            return externalCoordinates.y;
        }

        public int borderWidth(){
            return RWSConnector.DOTDIAM / 2;
        }

        /**
```

```
     * Return a (unique) textual representation of this node
     */
    public String toString(){
        return new String("[RWSNode:id=" + id + ";position=(" +
                          externalStartX() + "," + externalStartY() + "),(" +
                          externalEndX() + "," + externalEndY() + ")]");
    }

    /**
     * Sets the length of the nodes. Done from the template node.
     */
    public static void setDefaultNodeLength(int length){
        RWSNODELENGTH = length;
    }

    /**
     * Return the number of connectors in this compnent.
     */
    public int numberOfConnectors(){
        return connectors.length;
    }

    /**
     * Connects two connectors
     */
    private void connectConnectors ( RWSConnector mine, RWSConnector remote ){
        mine.neighbour = remote;
        remote.neighbour = mine;
    }

    public static int getCounter(){
        return counter;
    }

    /**
     * Update the global rwsNode counter
     */
    public static void setCounter (int c) {
        counter = c;
    }

    /**
     * Methods necessary for XMLEncoder
     */

    public void setComponentType (String type) {
        componentType = type;
    }

    public String getComponentType () {
        return componentType;
    }

    public int getStatus () {
```

```
        return status;
    }

    public void setCenterX (int x) {
        centerX = x;
    }

    public int getCenterX () {
        return centerX;
    }

    public void setCenterY (int y) {
        centerY = y;
    }

    public int getCenterY () {
        return centerY;
    }

    public void setEndP1X (int x) {
        endP1X = x;
    }

    public int getEndP1X () {
        return endP1X;
    }

    public void setEndP2X (int x) {
        endP2X = x;
    }

    public int getEndP2X () {
        return endP2X;
    }

    public void setEndP1Y (int y) {
        endP1Y = y;
    }

    public int getEndP1Y () {
        return endP1Y;
    }

    public void setEndP2Y (int y) {
        endP2Y = y;
    }

    public int getEndP2Y () {
        return endP2Y;
    }

    public void setId (int id) {
        this.id = id;
    }
```

```

    public int getId () {
        return id;
    }

    public void setTemplate (RWSNode template) {
        this.template = template;
    }

    public RWSNode getTemplate () {
        return template;
    }

    public void setExternalCoordinates (Rectangle rect) {
        externalCoordinates = rect;
    }

    public Rectangle getExternalCoordinates () {
        return externalCoordinates;
    }

    public void setConnectorX (int [] connectors) {
        connectorX = new int [connectors.length];
        for (int i = 0; i < connectors.length; i++) {
            connectorX [i] = connectors [i];
        }
    }

    public int [] getConnectorX () {
        return connectorX;
    }

    public void setConnectorY (int [] connectors) {
        connectorY = new int [connectors.length];
        for (int i = 0; i < connectors.length; i++) {
            connectorY [i] = connectors [i];
        }
    }

    public int [] getConnectorY () {
        return connectorY;
    }

    public void setNodeLength (int length) {
        nodeLength = length;
    }

    public int getNodeLength () {
        return nodeLength;
    }

    public void setConnectors (RWSConnector [] connectors) {
        if (connectors == null) {
            this.connectors = null;
```

```
            return;
        }
        this.connectors = new RWSConnector [connectors.length];
        for (int i = 0; i < connectors.length; i++) {
            this.connectors [i] = connectors [i];
            this.connectors [i].fixBounds ();
        }
    }

    public RWSConnector [] getConnectors () {
        return connectors;
    }

    /**
     * Given a RWSNode component n, generate its xml code.
     */
    public static Element createElement (Document doc, RWSNode n) {
        Element node, e;

        /**
         * The node component
         * <!ELEMENT  node   (#PCDATA | ... )*>
         */
        node = doc.createElement ("node");
        node.setAttribute ("id", Integer.toString (n.getId ()));
        if (! n.isTemplate)
            node.setAttribute ("templref",
                               Integer.toString (n.template.getId ()));

        /**
         * Necessary information
         * <!ELEMENT  info       EMPTY>
         * <!ATTLIST  info       componenttype  CDATA  #IMPLIED
         *            info       nodelength     CDATA  #REQUIRED
         *            info       status         CDATA  #REQUIRED>
         */
        e = doc.createElement ("info");
        if (n.getComponentType () != null)
            e.setAttribute ("componenttype", n.getComponentType ());
        e.setAttribute ("nodelength", Integer.toString (n.getNodeLength ()));
        e.setAttribute ("status", Integer.toString (n.getStatus ()));
        node.appendChild (e);

        /**
         * The coordinates and dimensions of the node
         * <!ELEMENT  placement          EMPTY>
         * <!ATTLIST  placement  x       CDATA  #REQUIRED
         *            placement  y       CDATA  #REQUIRED
         *            placement  width   CDATA  #REQUIRED
         *            placement  height  CDATA  #REQUIRED
         *            placement  centerX CDATA  #REQUIRED
         *            placement  centerY CDATA  #REQUIRED>
         */
        e = doc.createElement ("placement");
```

```
        Rectangle coords = n.getExternalCoordinates ();
        e.setAttribute ("x", Integer.toString (coords.x));
        e.setAttribute ("y", Integer.toString (coords.y));
        e.setAttribute ("width", Integer.toString (coords.width));
        e.setAttribute ("height", Integer.toString (coords.height));
        e.setAttribute ("centerX", Integer.toString (n.getCenterX ()));
        e.setAttribute ("centerY", Integer.toString (n.getCenterY ()));
        node.appendChild (e);

        /**
         * If this is an end element (only one connector)
         * <!ELEMENT  endcoordinates     EMPTY>
         */
        if (n.connectors.length == 1) {
            e = doc.createElement ("endcoordinates");
            e.setAttribute ("endp1x", Integer.toString (n.getEndP1X ()));
            e.setAttribute ("endp1y", Integer.toString (n.getEndP1Y ()));
            e.setAttribute ("endp2x", Integer.toString (n.getEndP2X ()));
            e.setAttribute ("endp2y", Integer.toString (n.getEndP2Y ()));
            node.appendChild (e);
        }

        /**
         * Walk through this node's connectors and add them
         */
        RWSConnector [] ctors = n.getConnectors ();
        for (int i = 0; i < n.connectors.length; i++) {
            node.appendChild (RWSConnector.createElement (doc,
                                                          ctors [i]));
        }

        return node;
    }
}
```

Listing 4: RWSConnector.java

```
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.event.MouseEvent;
import javax.swing.JPanel;
import java.awt.Graphics;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Rectangle;
import java.awt.Component;
import java.awt.Point;
import java.io.*;
import java.util.*;
import java.beans.XMLEncoder;
import java.beans.XMLDecoder;
import org.w3c.dom.*;
import org.xml.sax.*;

```

```
/**
 * RWSConnector objects are the interface nodes of the specification
 * language. They belong to RWSNodes and connect these to each other.
 */
public class RWSConnector extends JPanel implements MouseListener,
                                                    MouseMotionListener {

    final static int DOTDIAM = 8;
    private Point p;
    protected RWSNode node;

    protected CPNNode cpnInterface;
    protected RWSConnector neighbour;
    protected int xmlId;

    /* Used when traversing the graph */
    protected boolean mark = false;

    /* Used for moving a connector */
    boolean moving = false;

    boolean createElement = true;

    protected static RWSConnector selectedConnector, selectedTemplateConnector;

    /* Status */
    final protected static int UNUSED = 0;
    final protected static int USED = 1;
    final protected static int ACTIVE = 2;

    /* Mouse buttons */
    final protected static int MOUSE_LEFT = 0;
    final protected static int MOUSE_MIDDLE = 1;
    final protected static int MOUSE_RIGHT = 2;

    protected static RWSConnector hoveringConnector;

    protected static boolean connectNext = false;

    private int status = UNUSED;

    /* Array index in node.connectors */
    protected int index;

    /* Type decides which connectors we can connect to */
    protected int connectorType;

    /**
     * All the different rules for which connectors can connect to which.
     * Keys are Integer objects made from connectorTypes and values are
     * new hashmaps where the keys are valid connectorTypes. Values in this
     * last hashmap are the same as the keys, they exist solely for the purpose
     * of quick look-ups.
     */
```

```
    private static HashMap rules = new HashMap();


    /* Are we a template? */
    protected boolean isTemplate = true;

    /* Colors */
    final static Color COLOR_UNUSED = Color.red;
    final static Color COLOR_USED = Color.black;
    final static Color COLOR_ACTIVE = Color.blue;

    final private static Color [] currentColor =
        new Color [] { COLOR_UNUSED,
                       COLOR_USED,
                       COLOR_ACTIVE };

    /**
     * Empty constructor.
     */
    public RWSConnector () {
        setOpaque (false);
        addMouseListener (this);
        addMouseMotionListener (this);
    }

    /**
     * Constructor...
     */
    RWSConnector (int x, int y, RWSNode node, int index, boolean isTemplate) {
        setP (new Point (x, y));
        this.node = node;
        this.index = index;

        setOpaque (false);
        addMouseListener (this);
        addMouseMotionListener (this);

        this.setBounds (x - (DOTDIAM / 2), y - (DOTDIAM / 2),
                        DOTDIAM, DOTDIAM);

        if (selectedConnector != null)
            selectedConnector.setStatus (UNUSED);
        this.isTemplate = isTemplate;
    }

    public Dimension getPreferredSize () {
        return new Dimension (DOTDIAM, DOTDIAM);
    }

    /**
     * Check whether this connector can connecto to another
     * @param  RWSConnector  remote  the connector this is to connect to
     */
    protected boolean canConnectTo (RWSConnector remote){
```

```
        Integer connectorType1 = new Integer ( connectorType );
        Integer connectorType2 = new Integer ( remote.connectorType );

        if ( rules.containsKey ( connectorType1 ) ){
            HashMap h = (HashMap) rules.get ( connectorType1 );
            if ( h.containsKey ( connectorType2 ) )
                return true;
        }
        return false;
    }

    /**
     * Add a rule to allow two connector types to connect to each
     * others
     */
    protected static void addRule (RWSConnector conn1, RWSConnector conn2) {
        HashMap h;
        Integer cType = new Integer (conn1.connectorType);
        Integer canConnectTo = new Integer (conn2.connectorType);

        if (rules.containsKey (cType)) {
            h = (HashMap) rules.get (cType);
            if (! h.containsKey (canConnectTo))
                h.put (canConnectTo, canConnectTo);
        }
        else {
            h = new HashMap ();
            h.put (canConnectTo, canConnectTo);
            rules.put (cType, h);
        }

        if (rules.containsKey (canConnectTo)) {
            h = (HashMap) rules.get (canConnectTo);
            if (! h.containsKey (cType))
                h.put (cType , cType);
        }
        else {
            h = new HashMap ();
            h.put (cType , cType);
            rules.put (canConnectTo , h);
        }
    }

    /**
     * When a connectors status is changed from ACTIVE, we need
     * to find out whether to set it to USED or UNUSED.
     */

    public void unsetActive () {
        setStatus (isConnected () ? USED : UNUSED);
    }

    public boolean isActive(){
        return status == ACTIVE;
```

```
    }

    /**
     * Which mouse button was pressed?
     */
    private int getMouseButton(MouseEvent e){
        switch(e.getButton()){
        case MouseEvent.BUTTON1:
            return MOUSE_LEFT;
        case MouseEvent.BUTTON2:
            return MOUSE_MIDDLE;
        case MouseEvent.BUTTON3:
            return MOUSE_RIGHT;
        }
        return -1;
    }

    /**
     * Save the composition rules to xml for the specification .
     */
    public static Element createRulesElement (Document doc) {
        Element rule, rulesElement = null;
        try{
            rulesElement = doc.createElement ("rules");
            Iterator from = rules.keySet ().iterator ();
            while (from.hasNext ()) {
                Integer fromKey = (Integer) from.next ();
                HashMap h = (HashMap) rules.get (fromKey);
                Iterator to = h.keySet ().iterator ();
                while (to.hasNext ()) {
                    Integer toKey = (Integer) to.next ();
                    rule = doc.createElement ("rule");
                    rule.setAttribute ("from", fromKey.toString ());
                    rule.setAttribute ("to", toKey.toString ());
                    rulesElement.appendChild (rule);
                }
            }
        }
        catch (Exception e) {
            e.printStackTrace ();
        }

        return rulesElement;
    }

    /**
     * What happens when this connector is clicked?
     * Could be placed in mousePressed...
     *
     * Invariant 1: If a connector was previously selected it shall no longer
     *              be after this
     * Invariant 2: This connector shall be the selected connector after this
     *              unless it is the first in a "ring" which is closed
     * Invariant 3: Only a left click shall be able to select a connector
```

```
     */
    public void mouseClicked (MouseEvent e) {

        int mouseButton;

        /**
         * Firstly, this is a click on a connector. No node shall be selected.
         */
        if (RWSNode.selectedNode != null) {
            RWSNode.selectedNode.setStatus (RWSNode.INACTIVE);
            RWSNode.selectedNode.repaint();
            RWSNode.selectedNode = null;
        }

        /* is there a previously selected connector? */
        boolean existsPreviousConnector = selectedConnector != null;

        /* is there a previously selected node? */
        boolean existsPreviousNode = RWSNode.selectedNode != null;

        /* Return if I am the same as the previously selected connector */
        if (existsPreviousConnector &&
             selectedConnector == this)
            return;

        mouseButton = getMouseButton(e);

        /* We're not interested in center mouse clicks */
        if (mouseButton == MOUSE_MIDDLE)
            return;

        RWSNode previousNode = null;
        RWSConnector previousConnector = null;

        boolean addedRule = false;

        /**
         * This connector is part of a template
         */
        if (isTemplate) {
            /* Add a rule, DON'T SAVE THE PREVIOUS connector or node */
            if (RWSEditorFrame.action == RWSEditorFrame.CREATE_RULES &&
                 mouseButton == MOUSE_RIGHT) {
                addRule (selectedTemplateConnector, this);
                addedRule = true;

                previousNode = RWSNode.selectedTemplateNode;
                RWSNode.selectedTemplateNode = null;

                previousConnector = selectedTemplateConnector;
                selectedTemplateConnector = null;
            }
            /**
             * Select a new template connector, save the previous one
```

```
             * This also makes this connector's node the current
             * template node
             */
            else if ( mouseButton == MOUSE_LEFT ) {
                previousNode = RWSNode.selectedTemplateNode;
                RWSNode.selectedTemplateNode = node;
                previousConnector = selectedTemplateConnector;
                selectedTemplateConnector = this;
                setStatus (ACTIVE);
                repaint();
                RWSNode.setDefaultNodeLength ( node.nodeLength );
            }

            if (previousConnector != null && previousConnector != this){
                previousConnector.unsetActive();
                previousConnector.repaint();
            }

            /**
             * A template connector is clicked. If any connector was
             * previously selected, deselect this.
             */
            if ( existsPreviousConnector )
                selectedConnector.unsetActive();
            selectedConnector = null;
            RWSNode.selectedNode = null;

            if(mouseButton == MOUSE_RIGHT && !addedRule)
                RWSEditorFrame.popup.show(RWSEditorFrame.bg, e.getX() +
                                          externalCenterX(), e.getY() +
                                          externalCenterY());
        }

        /**
         * This connector is *NOT* part of a template
         */
        else {
            previousConnector = selectedConnector;
            previousNode = RWSNode.selectedNode;

            selectedConnector = this;
            RWSNode.selectedNode = node;

            /* We're not interested if it is a left click */
            switch (mouseButton){
            case MOUSE_LEFT:
                if (connectNext) {
                    if (previousConnector.connect (this)) {
                        previousConnector.moveConnector (
                            externalCenterX () -
                            previousConnector.externalCenterX (),
                            externalCenterY () -
                            previousConnector.externalCenterY ());
                    }
```

```
                    connectNext = false;
                }
                /**
                 * First, the case where we close a circle
                 * (The click is then on the "first" connector in the circle)
                 */
                else if ( existsPreviousConnector && /* There must be a
                                                   previous connector */
                     canConnectTo ( previousConnector ) && /* I can connect to
                                                              the previous
                                                              (first to last
                                                              in the circle) */
                     ! this.isConnected() && /* I'm available */
                     ! previousConnector.isConnected () ) { /* My peer is
                                                               available */
                    RWSEditor.frame.panel.repaint();
                    RWSEditor.frame.createNewNode(
                        previousConnector.externalCenterX(),
                        previousConnector.externalCenterY(),
                        externalCenterX(), externalCenterY(),
                        false, true);

                    /* Connect the first connector in the ring to the last */
                    selectedConnector.neighbour = this;
                    neighbour = selectedConnector;

                    /**
                     * Connect the third last connector
                     * in the ring to the second last
                     */
                    selectedConnector.node.connectors
                        [selectedTemplateConnector.index].neighbour =
                        previousConnector;
                    previousConnector.neighbour =
                        selectedConnector.node.connectors
                        [selectedTemplateConnector.index];

                    previousConnector.unsetActive();
                    selectedConnector.node.connectors
                        [selectedTemplateConnector.index].unsetActive();
                    selectedConnector.unsetActive();
                    unsetActive();

                    selectedConnector = null;
                    RWSNode.selectedNode = null;
                    node.setSelectedConnector();
                    return;
                }

                /* Second, no previously selected connector */
                else if ( ! existsPreviousConnector ){
                    if ( existsPreviousNode &&
                         node != previousNode )
                        previousNode.setStatus (RWSNode.INACTIVE);
```

```
                    RWSNode.selectedNode = node;
                    setStatus (ACTIVE);
                    if ( isConnected() ){
                        RWSNode.drawing = false;
                        RWSEditor.frame.panel.repaint();
                    }
                }

                /* Third, a connector was previously selected */
                else if ( existsPreviousConnector ){
                    if ( existsPreviousNode &&
                         node != previousNode )
                        previousNode.setStatus (RWSNode.INACTIVE);
                    previousConnector.unsetActive ();
                    setStatus (ACTIVE);
                    repaint();
                }

                /* We can DRAW! Set RWSNode.drawing */
                if ( ! isConnected() &&
                     canConnectTo ( selectedTemplateConnector ) ){
                    RWSNode.drawing = true;
                }
                break;

            case MOUSE_RIGHT:

                if ( ! addedRule )
                    RWSEditorFrame.popup.show(RWSEditorFrame.bg, e.getX() +
                                              externalCenterX(), e.getY() +
                                              externalCenterY());
                return;

            default:
                System.err.println("This␣is␣the␣default␣part␣of␣the␣switch,␣" +
                                   "where␣we␣should␣NOT␣be!");
            }
        }
    }

    /**
     * Connect this connector to another connector
     * @param  RWSConnector  toConnect  The peer to connect to
     * @since  1.18
     * @return boolean
     */
    protected boolean connect (RWSConnector toConnect) {
        if (canConnectTo (toConnect) &&    /* I can connect to toConnect */
            ! isConnected () &&       /* I'm available */
            ! toConnect.isConnected () ) { /* My peer is available */
            neighbour = toConnect;
            toConnect.neighbour = this;
            unsetActive ();
            toConnect.unsetActive ();
```

```
            return true;
        }
        return false;
    }

    /**
     * Interface method
     */
    public void mouseDragged (MouseEvent e) {
        moving = true;
    }

    /**
     * Interface method
     */
    public void mouseMoved (MouseEvent e) { }

    /**
     * Interface method
     */
    public void mousePressed (MouseEvent e) { }

    /**
     * Interface method
     */
    public void mouseReleased (MouseEvent e){
        if (moving) {
            int targetX, targetY;

            if (isConnected()) {
                if (hoveringConnector != null && hoveringConnector ==
                    neighbour)
                    return; /* We're above our own neighbour */
                else {
                    neighbour.neighbour = null;
                    neighbour.unsetActive ();
                    neighbour = null;
                    unsetActive ();
                }
            }

            if (hoveringConnector != null && hoveringConnector != this) {
                if (connect (hoveringConnector)) {
                    targetX = hoveringConnector.externalCenterX () -
                        externalCenterX ();
                    targetY = hoveringConnector.externalCenterY () -
                        externalCenterY ();
                }
                else {
                    System.err.println ("Could␣not␣connect");
                    return;
                }
            }
            else {
```

```
                targetX = e.getX () - (DOTDIAM / 2);
                targetY = e.getY () - (DOTDIAM / 2);
            }
            int xMove = targetX;
            int yMove = targetY;

            moveConnector (xMove, yMove);
            moving = false;
        }
    }

    /**
     * Move a connector and rescale the node
     * @param  int  x  How much to move the connector horizontally
     * @param  int  y  How much to move the connector vertically
     * @since  1.19
     * @return void
     */
    protected void moveConnector (int x, int y) {
        node.rescaleNode (x, y, index);
        setBounds(p.x - (DOTDIAM / 2), p.y - (DOTDIAM / 2),
                  DOTDIAM, DOTDIAM);
    }

    /**
     * Move this connector relative to the coordinates provided
     * @param  int  x  The horizontal distance to move the node
     * @param  int  y  The vertical distance to move the node
     * @since  1.15
     * @return void
     */
    protected void moveRelative (int x, int y) {
        p.x += x;
        p.y += y;
        setBounds (p.x - (DOTDIAM / 2), p.y - (DOTDIAM / 2),
                   DOTDIAM, DOTDIAM);
    }

    protected void fixBounds () {
        this.setBounds (p.x - (DOTDIAM / 2),
                        p.y - (DOTDIAM / 2),
                        DOTDIAM, DOTDIAM);
    }

    /**
     * Interface method
     */
    public void mouseEntered (MouseEvent e) {
        hoveringConnector = this;
    }

    /**
     * Interface method
     */
```

```
    public void mouseExited (MouseEvent e) {
        hoveringConnector = null;
    }

    /**
     * Paint this connector. Really simple.
     */
    public void paintComponent(Graphics g){
        g.setColor (currentColor [status]);
        g.fillOval (0, 0, DOTDIAM, DOTDIAM);
    }

    /**
     * Return the coordinates of this connector's center point
     */
    public int externalCenterX (){
        return node.externalStartX() + p.x;
    }

    /**
     * Return the coordinates of this connector's center point
     */
    public int externalCenterY (){
        return node.externalStartY() + p.y;
    }

    /**
     * Return the coordinates of this connector's center point
     * where (0, 0) is the starting point of the connector
     */
    public int internalCenterX (){
        return p.x;
    }

    /**
     * Return the coordinates of this connector's center point
     * where (0, 0) is the starting point of the connector
     */
    public int internalCenterY (){
        return p.y;
    }

    /**
     * Set which node this connector "belongs" to.
     */
    protected void setNode (RWSNode node){
        this.node = node;
    }

    /**
     * Static method to return the index of the selected template
     * node's selected connector.
     */
    protected static int selectedTemplateConnectorIndex(){
```

```
        if(selectedTemplateConnector != null)
            return selectedTemplateConnector.index;
        return 0;
    }

    /**
     * Changes this connector's type. Affects which connectors
     * it can connect to.
     */
    protected void setConnectorType(int type){
        connectorType = type;
    }

    /**
     * Change this connector's status.
     */
    protected void setStatus(int status){
        this.status = status;
    }

    /**
     * Are we connected?
     */
    protected boolean isConnected () {
        return neighbour != null;
    }

    /**
     * Are we connected?
     * @param String inter The id number of the interface to connect to
     * @param String comp  The component (actually the name of the XML file)
     */
    protected void addCPNInterface ( String inter , String comp ) {
        if ( isTemplate )
            node.cpnComponentType = comp;
        else
            node.template.cpnComponentType = comp;
        CPNNode n;
        Place n1 = null;
        HashMap cpnNodes = ( HashMap ) XMLUtils.cpnComponents.get ( comp );
        if ( cpnNodes == null ) {
            return;
        }
        Iterator it = cpnNodes.keySet ().iterator ();

        while (it.hasNext()) {
            String key = (String) it.next ();
            n = (CPNNode) cpnNodes.get (key);
            if (n instanceof Place){
                Place pl = (Place) n;
                if (pl.getId ().equals (inter)) {
                    pl.setInterface (true);
                    n1 = pl;
                }
```

```
                else
                    System.err.println ();
            }
        }
        cpnInterface = n1;
    }

    /**
     * Return the cpn node.
     */
    protected CPNNode getCPNInterface(){
        if (! isTemplate)
            return node.template.connectors [index].cpnInterface;
        else
            return cpnInterface;
    }

    /**
     * Return what identifies this connector
     */
    public String toString(){
        return "[RWSConnector:id=" + node.id + "." + index + ";type=" +
            connectorType + ";position=(" + externalCenterX() + "," +
            externalCenterY() + ")]";
    }

    /**
     * Methods necessary for XMLEncoder
     */

    public void setP (Point p) {
        this.p = p;
    }

    public Point getP () {
        return p;
    }

    public RWSNode getNode () {
        return node;
    }

    public void setCpnInterface (CPNNode iface) {
        cpnInterface = iface;
    }

    public String getCpnInterfaceId () {
        return cpnInterface.getId ();
    }

    public void setNeighbour (RWSConnector neighbour) {
        this.neighbour = neighbour;
    }

```

```
    public RWSConnector getNeighbour () {
        return neighbour;
    }

    public void setXmlId (int id) {
        xmlId = id;
    }

    public int getXmlId () {
        return xmlId;
    }

    public int getStatus () {
        return status;
    }

    public void setIndex (int index) {
        this.index = index;
    }

    public int getIndex () {
        return index;
    }

    public int getConnectorType () {
        return connectorType;
    }

    public static void setRules (HashMap rules) {
        RWSConnector.rules = rules;
    }

    /*
     * Create the xml code for this connector.
     */
    public static Element createElement (Document doc, RWSConnector c) {
        Element conn, e;

        /**
         * The connector component
         * <!ELEMENT  connector   (#PCDATA | ... )*>
         * <!ATTLIST  connector   index       CDATA  #REQUIRED
         *            connector   noderef     IDREF  #REQUIRED
         *            connector   istemplate  CDATA  #REQUIRED>
         */
        conn = doc.createElement ("connector");
        conn.setAttribute ("noderef",
                           Integer.toString (c.getNode ().getId ()));
        conn.setAttribute ("index", Integer.toString (c.getIndex ()));
        conn.setAttribute ("istemplate", c.isTemplate ? "true" : "false");

        /**
         * Position
         * <!ELEMENT  pos    EMPTY>
```

```
         * <!ATTLIST  pos    x          CDATA  #REQUIRED
         *            pos    y          CDATA  #REQUIRED>
         */
        e = doc.createElement ("pos");
        e.setAttribute ("x", Integer.toString (c.getP ().x));
        e.setAttribute ("y", Integer.toString (c.getP ().y));
        conn.appendChild (e);

        /**
         * Neighbour ("optional")
         * <!ELEMENT  neighbour    EMPTY>
         * <!ATTLIST  neighbour    node   IDREF  #REQUIRED
         *            neighbour    index  CDATA  #REQUIRED>
         */
        if (c.getNeighbour () != null) {
            e = doc.createElement ("neighbour");
            e.setAttribute ("node",
                            Integer.toString (
                                c.getNeighbour ().getNode ().getId ()));
            e.setAttribute ("index",
                            Integer.toString (
                                c.getNeighbour ().getIndex ()));
            conn.appendChild (e);
        }

        /**
         * Vital information
         * <!ELEMENT  info    EMPTY>
         * <!ATTLIST  info    status        CDATA  #REQUIRED
         *            info    connectortype CDATA  #REQUIRED>
         */
        e = doc.createElement ("info");
        e.setAttribute ("status", Integer.toString (c.getStatus ()));
        e.setAttribute ("connectortype",
                        Integer.toString (c.getConnectorType ()));
        conn.appendChild (e);

        return conn;
    }
}
```

Listing 5: CPNNode.java

```
import java.util.Vector;
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.w3c.dom.*;
import java.io.*;

/**
 * Superclass for internal representation of Petri Net components
 */
class CPNNode implements Serializable {

```

```java
    private String id;
    Vector neighbours;
    int xmlId = 0;
    Node xmlnode;

    public CPNNode (String id, Node n) {
        this.id = id;
        xmlnode = n;
    }

    /**
     * Add a neighbour component
     */
    public void addNeighbour (CPNNode n) {
        if (neighbours == null)
            neighbours = new Vector ();
        /* First, add the neighbour */
        if (! hasNeighbour (n))
            neighbours.add (n);
        /**
         * Since this is also a neighbour of n,
         * add this to n's neighbours
         */
        if (! n.hasNeighbour (this))
            n.addNeighbour (this);
    }

    /**
     * Return whether we have a specified neighbour
     */
    public boolean hasNeighbour (CPNNode n) {
        if (neighbours != null)
            return neighbours.contains (n);
        return false;
    }

    /**
     * Set this component's id
     */
    public void setId (String id) {
        this.id = id;
    }

    /**
     * Return this component's id
     */
    public String getId () {
        return id;
    }
}
```

Listing 6: Place.java

```java
import javax.xml.parsers.*;
```

```java
import org.xml.sax.*;
import org.w3c.dom.*;

/**
 * Class for internal representation of Petri Net Places
 */
class Place extends CPNNode {

    private String name;
    private boolean isInterface = false;

    Place (String id, Node n) {
        super (id, n);
    }

    /**
     * Set whether this is as interface place
     */
    void setInterface (boolean inter) {
        isInterface = inter;
    }

    /**
     * Set this place's name
     */
    void setName (String n) {
        name = n;
    }

    /**
     * Return this place's name
     */
    String getName () {
        return name;
    }
}
```

Listing 7: Arc.java

```java
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.w3c.dom.*;

/**
 * Class for internal representation of Petri Net arcs
 */
class Arc extends CPNNode{

    String placeend;
    String transend;
    int xmlPlaceend;
    int xmlTransend;

    Arc(String id, Node node){
```

```java
        super(id, node);
    }

    /**
     * Methods to set and get the places and transitions on the ends
     * of this arc
     */

    void setPlaceend (String s){
        placeend = s;
    }

    void setTransend (String t){
        transend = t;
    }

    String getPlaceend (){
        return placeend;
    }

    String getTransend (){
        return transend;
    }
}
```

Listing 8: Transition.java

```java
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.w3c.dom.*;

/**
 * Class for internal representation of Petri Net Transitions
 */
class Transition extends CPNNode {

    String name;

    Transition (String id, Node n) {
        super (id, n);
    }

    /**
     * Set this transition's name
     */
    void setName (String n) {
        name = n;
    }

    /**
     * Return this transition's name
     */
    String getName () {
        return name;
```

```
    }
}
```

Listing 9: XMLUtils.java

```
import java.awt.*;
import java.io.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.sax.*;
import javax.xml.transform.stream.*;
import org.xml.sax.*;
import org.w3c.dom.*;
import java.util.*;
import javax.swing.*;

/*
* The XMLUtils class handles reading and generating of XML files.
* - Read CPN components according to Design/CPN's DTD,
* - Write the generated CPN implementation to file according to Design/CPN's DTD.
* - Write the specification to file.
* - Read the specification from file.
*/

class XMLUtils{

    /**
     * RWSConnector.mark and RWSConnector.createElement are now used in
     * conjunction with currentMark to make it possible to save to CPN XML
     * more than once.
     */

    boolean createPageElement = true;

    Element ele;
    Element cpnetEl;
    Document document;

    HashMap xmlNodes, cpnNodes;

    /* These variables are used when opening a file */
    private HashMap connectorsToConnect, allConnectors, templateNodes;
    private int maxId = 0;

    /* Every CPN node needs a unique id */
    int lopeid = 100;

    private boolean currentMark = false;
    private String fileName;

    /**
     * HashMap cpnComponents contains a hashMap for each CPN
     * components. The key is the component name and element of the
```

```
     * hashMap is: Places, Transitions and arcs.
     */
    static HashMap cpnComponents = new HashMap();

    /**
     * Read the XML file for a CPN component and create a hashMap of
     * xmlNodes and a hashMap of cpnNodes.
     */
    void readXML(String filepath, String filename){

        File file = new File(filepath);

        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        try{

            DocumentBuilder builder = factory.newDocumentBuilder();
            document = builder.parse (file);

            xmlNodes = new HashMap();
            cpnNodes = new HashMap();

            NodeList nl;

            nl = document.getElementsByTagName("page");
            Node n = nl.item(0);

            nl = n.getChildNodes();
            String nodeName;
            for(int i=0;i<nl.getLength();i++){
                n = nl.item(i);
                nodeName = n.getNodeName();
                if(nodeName == "place" || nodeName == "trans" ||
                   nodeName == "arc"){
                    xmlNodes.put(getID(n), n);
                }
            }

            nl = document.getElementsByTagName("arc");
            Arc a;
            Place p;
            Transition t;
            String id;
            Node tmpNode;
            for(int i=0;i<nl.getLength();i++){
                n = nl.item(i);
                id = getID(n);
                a = new Arc(id,n);
                cpnNodes.put(id, a);

                // Get a's adjacent place (still xml node)
                tmpNode = getArcEndPoint(n, xmlNodes, "placeend");
                String tmpNodeName = findName(tmpNode);

                // Do we have n as a CPNNode yet?
```

```
                id = getNodeAttribute(tmpNode, "id");
                a.setPlaceend(id);
                if(cpnNodes.containsKey(id))
                    // Yes, retrieve it from the hashmap cpnNodes
                    p = (Place) cpnNodes.get(id);
                else{
                    // No, create a new Place...
                    p = new Place(id, tmpNode);
                    p.setName(tmpNodeName);
                    // ...and put it into cpnNodes
                    cpnNodes.put(id, p);
                }

                // Same thing with transitions
                tmpNode = getArcEndPoint(n, xmlNodes, "transend");
                tmpNodeName = findName(tmpNode);
                id = getNodeAttribute(tmpNode, "id");
                a.setTransend(id);
                if(cpnNodes.containsKey(id))
                    t = (Transition) cpnNodes.get(id);
                else{
                    t = new Transition(id,tmpNode);
                    t.setName(tmpNodeName);
                    cpnNodes.put(id, t);
                }

                a.addNeighbour(p);
                a.addNeighbour(t);
            }

            Iterator it = xmlNodes.keySet().iterator();

            while(it.hasNext()){
                String key = (String) it.next();
                Node placenode = (Node) xmlNodes.get(key);

                if(placenode.getNodeName().equals("place")){
                    if(!cpnNodes.containsKey(key)){
                        p = new Place(key, placenode);
                        cpnNodes.put(key,p);
                    }
                }
            }

            cpnComponents.put (filename, cpnNodes);

        } catch (SAXException sxe) {
            // Error generated during parsing
            Exception  x = sxe;
            if (sxe.getException() != null)
                x = sxe.getException();
            x.printStackTrace();

        } catch (ParserConfigurationException pce) {
```

```
            // Parser with specified options can't be built
            pce.printStackTrace();

        } catch (IOException ioe) {
            // I/O error
            ioe.printStackTrace();
        }
    }

    /**
     * Write CPN XML file
     */
    void outputXMLFile (String outfile) {
        try{
            PrintWriter out = new PrintWriter(new FileWriter(outfile));

            NodeList nl = cpnetEl.getChildNodes();
            out.println("<"+cpnetEl.getNodeName()+">");

            Node n1 = nl.item(0);
            String pageId = getNodeAttribute(n1,"id");
            out.println("<" + n1.getNodeName() +"␣"+ "id=\""+pageId+"\"" +">");
            nl = n1.getChildNodes();
            n1.normalize ();

            for(int i = 0; i<nl.getLength(); i++){
                if (nl.item (i).hasChildNodes ()) {
                    NodeList nl2 = nl.item (i).getChildNodes ();
                    out.print ("<" + nl.item (i).getNodeName ());
                    NamedNodeMap map = nl.item (i).getAttributes ();
                    for (int j = 0; j < map.getLength (); j++) {
                        out.print ("␣" + map.item (j).getNodeName () + "=\"" +
                                   map.item (j).getNodeValue () + "\"");
                    }
                    out.println(">");
                    for(int j = 0; j < nl2.getLength(); j++){
                        if (nl2.item (j).hasChildNodes ()) {
                            NodeList nl3 = nl2.item (j).getChildNodes ();
                            out.print ("␣␣<" + nl2.item (j).getNodeName ());
                            map = nl2.item (j).getAttributes ();
                            for (int k = 0; k < map.getLength (); k++) {
                                out.print ("␣" + map.item (k).getNodeName () +
                                           "=\"" + map.item (k).getNodeValue ()
                                           + "\"");
                            }
                            out.println(">");
                            for(int k = 0; k < nl3.getLength(); k++){
                                out.println("␣␣␣␣" + nl3.item (k));
                            }
                            out.println ("␣␣</" + nl2.item (j).getNodeName ()
                                        + ">");
                        }
                        else
                            out.println("␣␣" + nl2.item (j));
```

```
                    }
                    out.println ("</" + nl.item (i).getNodeName () + ">");
                }
                else
                    out.println(nl.item (i));
            }

            out.println("</"+n1.getNodeName() +">");
            out.println("</"+cpnetEl.getNodeName()+">");
            out.close();
        } catch (IOException ioe) {
            // I/O error
            ioe.printStackTrace();
        }
    }

    /**
    * Methods for getting the ID, the value of a given attribute and
    * a specific child node of XML .
    */
    String getID(Node n){
        NamedNodeMap nm = n.getAttributes();
        for(int i=0;i<nm.getLength();i++){
            n = nm.item(i);
            if(n.getNodeName() == "id")
                return n.getNodeValue();
        }
        return null;
    }

    String getNodeAttribute(Node n, String key){
        NamedNodeMap nm = n.getAttributes();
        for(int i=0;i<nm.getLength();i++){
            n = nm.item(i);
            if(n.getNodeName() == key)
                return n.getNodeValue();
        }
        return null;
    }

    Node getNodeChild(Node n, String key){
        NodeList nl = n.getChildNodes();
        for(int i=0;i<nl.getLength();i++){
            n = nl.item(i);
            if(n.getNodeName() == key)
                return n;
        }
        return null;
    }

    /**
     * Get the place or transition that is on one end of an arc
     * @param  Node     a         a specific xml arc node
     * @param  HashMap  xmlNodes  hash containing all xml nodes
```

```
     * @param  Strinng  type      placeend / transend
     */
    Node getArcEndPoint(Node a, HashMap xmlNodes, String type){
        return (Node) xmlNodes.get(getNodeAttribute(getNodeChild(a, type),
                                                         "idref"));
    }

    /* Find name of place or transition, if they exist */
    String findName(Node xmlNode){
        String name = "";
        if(!xmlNode.getNodeName().equals("arc")){
            Node childnode = getNodeChild(xmlNode,"name");
            if(childnode !=null){
                childnode = getNodeChild(childnode,"text");
                if(childnode.hasChildNodes()){
                    NodeList namelist  = childnode.getChildNodes();
                    for(int i=0;i<namelist.getLength(); i++){
                        name = namelist.item(i).getNodeValue();
                    }
                }
            }
        }
        return name;
    }

    /* This method is just for debugging */
    void printout (RWSNode r) {
        System.out.println("jeg␣har␣(RWS)id␣:" + r.id);

        if ( r .connectors [0].neighbour != null)
            System.out.println("min␣nabo␣[0]␣har␣(RWS)␣id␣:" +
                               r.connectors [0].neighbour.node.id);
        else
            System.out.println("jeg␣har␣ingen␣nabo␣[0]");

        if ( r.connectors [1].neighbour != null)
            System.out.println("min␣nabo␣[1]␣har␣(RWS)␣id␣:" +
                               r.connectors [1].neighbour.node.id);
        else
            System.out.println("jeg␣har␣ingen␣nabo␣[1]");

        if(r.connectors [0].cpnInterface != null &&
           r.connectors [1].cpnInterface != null){
            Place p = (Place) r.connectors [0].cpnInterface;
            System.out.println("min␣inter␣[0]␣har␣id␣:" +
                               p.getId() + "og␣navn:␣" + p.getName());
            p = (Place) r.connectors [1].cpnInterface;
            System.out.println("min␣inter␣[1]␣har␣␣id␣:" +
                               p.getId() + "og␣navn:␣" + p.getName() + "\n");
        }

        if(r.connectors [1].neighbour != null)
            printout ( r.connectors [1].neighbour.node );
    }
```

```

    /**
     * Given a root RWSNode, generate the cpn XML file. This method has some
     * subrutines.
     * Method PrintXML() and creatNewElement() is the main methods /
     * responsibility for generating the CPN xml file from the specification.
     */
    void initPrintXml (RWSNode n, String filename) {
        fileName = filename;

        assignXmlID (n);
        printXML (n);
        cpnetEl.appendChild (ele);
        outputXMLFile (filename);
        currentMark = ! currentMark;
    }

    /**
     * Recursively assign each connector of RWSNodes with a xml_id.
     * This is necessary for Design/CPN-.
     */
    void assignXmlID(RWSNode node){
        for(int i=0;i<node.connectors.length;i++){
            if ( node.connectors [i].mark == currentMark ){
                RWSConnector connector = node.connectors [i];
                connector.mark = ! currentMark;
                connector.xmlId = lopeid++;
                if(connector.neighbour != null){
                    connector.neighbour.xmlId = connector.xmlId;
                    connector.neighbour.mark = ! currentMark;
                    assignXmlID( connector.neighbour.node );
                }
            }
        }
    }

    /**
     * For each RWSNode, retrive its underlying CPN model
     * and calls createNewElement ()
     * to create XML code for this CPN model.
     */
    void printXML (RWSNode n){

        RWSNode neighb = null;
        RWSNode tmp = n;
        if(tmp != null && tmp.mark == currentMark){
            tmp.mark = ! currentMark;

            for(int i=0; i<tmp.connectors.length;i++){
                if (tmp.connectors [i].getCPNInterface () == null)
                    return;

                tmp.connectors [i].getCPNInterface ().xmlId =
                    tmp.connectors [i].xmlId;
```

```

            if ( tmp.connectors [i].createElement != currentMark ) {
                /**
                 * This connector has not been treated yet. Therefore,
                 * we retrieve the underlying CPN node (CPN place) and write it.
                 * That means that this connector is an interface.
                 * We use currentMark for reusability.
                 */
                createNewElement ( tmp.connectors [i].getCPNInterface(),
                                   tmp.connectors [i], tmp );
                tmp.connectors [i].createElement = currentMark;
                if ( tmp.connectors [i].neighbour != null ){
                    /**
                     *This connector's neighbour shall not be treated,
                     * as it is the same as this connector in the CPNet.
                     */
                    tmp.connectors [i].neighbour.createElement =
                        currentMark;
                }
            }
        }

        String compType = (tmp.isTemplate) ? tmp.cpnComponentType :
            tmp.template.cpnComponentType;
        HashMap cpnComp = (HashMap) cpnComponents.get (compType);
        Iterator it = cpnComp.keySet().iterator();

        while(it.hasNext()){
            String key = (String)it.next();
            CPNNode cpn = (CPNNode) cpnComp.get(key);
            boolean isInterface=false;
            for(int i=0; i<tmp.connectors.length;i++){
                if ( tmp.connectors [i].getCPNInterface() == cpn)
                    isInterface = true;
            }

            /**
             * write xml code for CPN places
             * that are not interfaces, and transitions.
             */
            if(!isInterface){
                if(cpn instanceof Place || cpn instanceof Transition){
                    cpn.xmlId = lopeid++;
                    createNewElement(cpn, null, tmp);
                }
            }
        }

        Iterator iter = cpnComp.keySet().iterator();
        /* write xml code for arcs*/
        while(iter.hasNext()){
            String key =(String)iter.next();
            CPNNode cpnn = (CPNNode)cpnComp.get(key);
            if(cpnn instanceof Arc){
```

```
                    Arc a = (Arc)cpnn;
                    a.xmlId = lopeid++;
                    String p = a.getPlaceend();
                    Place pl = (Place) cpnComp.get(p);
                    a.xmlPlaceend = pl.xmlId;

                    String t = a.getTransend();
                    Transition tran = (Transition) cpnComp.get(t);
                    a.xmlTransend = tran.xmlId;
                    createNewElement(cpnn, null, tmp);
                }
            }

            for(int i=0; i<tmp.connectors.length; i++){
                if ( tmp.connectors [i].neighbour != null &&
                     tmp.connectors [i].neighbour.node.mark == currentMark ){
                    printXML ( tmp.connectors [i].neighbour.node );

                }
            }
        }
    }

    /**
     * This method generates the necessary xml code for each cpn component.
     * @param  CPNNode      c   The CPN node to be output
     * @param  RWSConnector rc  The corresponding RWSConnector
     * @param  RWSNode      rn  The corresponding RWSNode
     * @since  0
     * @return void
     */
    void createNewElement(CPNNode c, RWSConnector rc, RWSNode rn){
        Element child;
        String nodeType = "";
        String color = "";

        Element n =null;
        String orientation = "";
        boolean setPosition = false;
        int pX, pY;
        Dimension size = RWSEditor.frame.panel.getPreferredSize();

        /**
         * OK. When we want to calculate where to place this element,
         * we need some info. If this is an interface, rc != null and c
         * is a Place. Then, if c is the first "part" of the CPN component
         * to be processed, we need to determine where to put it. Otherwise,
         * we have to place c relative to the first one.
         */

        if (rc != null && rc.node.positionMark == currentMark) {

            rc.node.positionMark = ! currentMark;
            setPosition = true;
```

```
            rc.node.xmlX = (rc.externalCenterX () * 3) -
                            RWSEditor.frame.meanX;
            rc.node.xmlY = ((size.height - rc.externalCenterY ()) * 3) -
                            RWSEditor.frame.meanY;

            pX = rc.node.xmlX;
            pY = rc.node.xmlY;
            Node posNode = getNodeChild (c.xmlnode, "posattr");
            if (posNode == null) {
                rc.node.relX = 0;
                rc.node.relY = 0;
            }
            else {
                try {
                    rc.node.relX = (int) Double.parseDouble (
                        getNodeAttribute (posNode, "x"));
                    rc.node.relY = (int) Double.parseDouble (
                        getNodeAttribute (posNode, "y"));
                }
                catch (NumberFormatException ex) {
                    ex.printStackTrace ();
                }
            }
        }
        else {
            Node posNode = getNodeChild (c.xmlnode, "posattr");
            if (posNode != null){
                try{
                    pX = rn.xmlX + (int) Double.parseDouble (
                        getNodeAttribute (posNode, "x")) - rn.relX;
                    pY = rn.xmlY + (int) Double.parseDouble (
                        getNodeAttribute (posNode, "y")) - rn.relY;
                }
                catch (NumberFormatException ex) {
                    ex.printStackTrace ();
                    pX = rn.xmlX - rn.relX;
                    pY = rn.xmlY - rn.relY;
                }
            }
            else{
                pX = rn.xmlX - rn.relX;
                pY = rn.xmlY - rn.relY;
            }
        }
        /* xml code for the page element */
        if (createPageElement) {
            NodeList nl = document.getElementsByTagName("page");
            Node nn = nl.item(0);

            cpnetEl = document.createElement("cpnet");

            ele = document.createElement("page");
            ele.setAttribute ("id", "id"+lopeid++);

```

```
            child = document.createElement("pageattr");
            child.setAttribute("name",fileName);
            child.setAttribute("number",
                            getNodeAttribute (getNodeChild(nn, "pageattr"),
                                              "number"));
            child.setAttribute("visbor",
                            getNodeAttribute(getNodeChild(nn, "pageattr"),
                                             "visbor"));
            child.setAttribute("palette",
                            getNodeAttribute(getNodeChild(nn, "pageattr"),
                                             "palette"));
            ele.appendChild(child);

            child = document.createElement("mult");
            child.setAttribute("insts",
                            getNodeAttribute(getNodeChild(nn, "mult"),
                                             "insts"));
            ele.appendChild(child);

            child = document.createElement("winattr");
            child.setAttribute("open",
                            getNodeAttribute(getNodeChild(nn, "winattr"),
                                             "open"));
            child.setAttribute("width",
                            getNodeAttribute(getNodeChild(nn, "winattr"),
                                             "width"));
            child.setAttribute("height",
                            getNodeAttribute(getNodeChild(nn, "winattr"),
                                             "height"));
            child.setAttribute("xpos",
                            getNodeAttribute(getNodeChild(nn, "winattr"),
                                             "xpos"));
            child.setAttribute("ypos",
                            getNodeAttribute(getNodeChild(nn, "winattr"),
                                             "ypos"));
            ele.appendChild(child);

            child = document.createElement("lineattr");
            child.setAttribute("type",
                            getNodeAttribute(getNodeChild(nn, "lineattr"),
                                             "type"));
            child.setAttribute("thick",
                            getNodeAttribute(getNodeChild(nn, "lineattr"),
                                             "thick"));
            child.setAttribute("colour",
                            getNodeAttribute(getNodeChild(nn, "lineattr"),
                                             "colour"));
            ele.appendChild(child);

            child = document.createElement("posattr");
            child.setAttribute("x",
                            getNodeAttribute(getNodeChild(nn, "posattr"),
                                             "x"));
            child.setAttribute("y",
```

```
                            getNodeAttribute(getNodeChild(nn, "posattr"),
                                             "y"));
            ele.appendChild(child);

            child = document.createElement("box");
            child.setAttribute("h",
                            getNodeAttribute(getNodeChild(nn, "box"),
                                             "h"));
            child.setAttribute("w",
                            getNodeAttribute(getNodeChild(nn, "box"),
                                             "w"));
            ele.appendChild(child);

            createPageElement = false;
        }

        /* Find out if this CPNNode is a Place, Transition or Arc*/
        if(c instanceof Place){
            nodeType = "place";

            Node node = getNodeChild (c.xmlnode, "type");
            if (node != null){
                node = getNodeChild (node,"text");
                if (node.hasChildNodes ()) {
                    NodeList namelist  = node.getChildNodes ();
                    for (int i = 0;i < namelist.getLength (); i++) {
                        color = namelist.item (i).getNodeValue ();
                    }
                }
            }
        }

        else if(c instanceof Transition){
            nodeType = "trans";
        }

        else if(c instanceof Arc){
            nodeType = "arc";
            orientation = getNodeAttribute(c.xmlnode, "orientation");
        }


        int id = c.xmlId;
        n = document.createElement(nodeType);
        n.setAttribute("id", "id"+id);

        /* Generate necessary XML code for an arc */
        if(c instanceof Arc){
            n.setAttribute("orientation", orientation);
            child = document.createElement("connattr");
            child.setAttribute("hdwidth",
                            getNodeAttribute(getNodeChild(c.xmlnode,
                                                          "connattr"),
                                             "hdwidth"));
```

```
            child.setAttribute("hdheight",
                         getNodeAttribute(getNodeChild(c.xmlnode,
                                                      "connattr"),
                                            "hdheight"));
            child.setAttribute("txtwidth",
                         getNodeAttribute(getNodeChild(c.xmlnode,
                                                      "connattr"),
                                            "txtwidth"));
            child.setAttribute("txtheight",
                         getNodeAttribute(getNodeChild(c.xmlnode,
                                                      "connattr"),
                                            "txtheight"));
            n.appendChild(child);
        }

        child = document.createElement("flags");
        child.setAttribute("visible", "true");
        n.appendChild(child);

        child = document.createElement("lineattr");
        child.setAttribute("thick", "1");
        child.setAttribute("colour", "black");
        n.appendChild(child);

        child = document.createElement("textattr");
        child.setAttribute("size", "10");
        child.setAttribute("colour", "black");
        n.appendChild(child);

        child = document.createElement("posattr");
        child.setAttribute("x", Integer.toString(pX));
        child.setAttribute("y", Integer.toString(pY));
        n.appendChild(child);

        /* Generate necessary XML code for a Place */
        if (c instanceof Place){
            child = document.createElement("ellipse");
            child.setAttribute("h",
                         getNodeAttribute(getNodeChild(c.xmlnode,
                                                      "ellipse"),
                                            "h"));
            child.setAttribute("w",
                         getNodeAttribute(getNodeChild(c.xmlnode,
                                                      "ellipse"),
                                            "w"));
            n.appendChild(child);

            child = document.createElement("name");
            child.setAttribute("id", "id"+lopeid++);
            Element childOFChild = document.createElement("posattr");
            childOFChild.setAttribute("x", Integer.toString(pX));
            childOFChild.setAttribute("y", Integer.toString(pY));
            child.appendChild(childOFChild);

```

```
            Node node = getNodeChild(c.xmlnode, "name");
            String placeName = "";
            if(node != null){
                node = getNodeChild(node, "text");
                if(node.hasChildNodes()){
                    NodeList namelist  = node.getChildNodes();
                    for(int i=0;i<namelist.getLength(); i++){
                        placeName = namelist.item(i).getNodeValue();
                    }
                }
            }

            childOFChild = document.createElement("text");

            Text name = document.createTextNode(placeName+rn.id);

            childOFChild.appendChild(name);
            child.appendChild(childOFChild);

            n.appendChild(child);

            child = document.createElement("type");
            child.setAttribute("id", "id"+lopeid++);
            childOFChild = document.createElement("lineattr");
            childOFChild.setAttribute("colour", "black");
            child.appendChild(childOFChild);

            childOFChild = document.createElement("posattr");
            childOFChild.setAttribute("x", Integer.toString(pX));
            childOFChild.setAttribute("y", Integer.toString(pY+10));
            child.appendChild(childOFChild);

            childOFChild = document.createElement("textattr");
            childOFChild.setAttribute("colour", "black");
            child.appendChild(childOFChild);

            childOFChild = document.createElement("text");
            Text t = document. createTextNode(color);
            childOFChild.appendChild(t);
            child.appendChild(childOFChild);

            n.appendChild(child);

            node = getNodeChild(c.xmlnode, "initmark");
            String placemark = "";
            if(node != null){
                node = getNodeChild(node, "text");
                if(node.hasChildNodes()){
                    NodeList namelist  = node.getChildNodes();
                    for(int i=0;i<namelist.getLength(); i++){
                        placemark = namelist.item(i).getNodeValue();

                    }
                }
```

```

                    child = document.createElement("initmark");
                    child.setAttribute("id", "id"+lopeid++);
                    childOFChild = document.createElement("lineattr");
                    childOFChild.setAttribute("colour", "black");
                    child.appendChild(childOFChild);

                    childOFChild = document.createElement("posattr");
                    childOFChild.setAttribute("x", Integer.toString(pX));
                    childOFChild.setAttribute("y", Integer.toString(pY-10));
                    child.appendChild(childOFChild);

                    childOFChild = document.createElement("textattr");
                    childOFChild.setAttribute("colour", "black");
                    child.appendChild(childOFChild);

                    childOFChild = document.createElement ("text");
                    t = document.createTextNode (placemark);
                    childOFChild.appendChild (t);
                    child.appendChild (childOFChild);
                    n.appendChild (child);
                }
            }

            /* Generate necessary XML code for a Transition */
            if(c instanceof Transition){
                Node node = getNodeChild(c.xmlnode,"box");
                String height = "15";
                String width = "15";

                if(node != null){
                    height = getNodeAttribute(node, "h");
                    width = getNodeAttribute(node, "w");
                }

                child = document.createElement("box");
                child.setAttribute("h", height);
                child.setAttribute("w", width);
                n.appendChild(child);

                child = document.createElement("name");
                child.setAttribute("id", "id"+lopeid++);
                Element childOFChild = document.createElement("posattr");
                childOFChild.setAttribute("x", Integer.toString(pX));
                childOFChild.setAttribute("y", Integer.toString(pY));
                child.appendChild(childOFChild);

                node = getNodeChild(c.xmlnode, "name");
                String tranName = "";

                if(node != null){
                    node = getNodeChild(node, "text");
                    if(node.hasChildNodes()){
                        NodeList namelist  = node.getChildNodes();
```

```
                    for(int i=0;i<namelist.getLength(); i++){
                        tranName = namelist.item(i).getNodeValue();
                    }
                }
            }

            childOFChild = document.createElement("text");
            Text t = document. createTextNode(tranName+rn.id);
            childOFChild.appendChild(t);
            child.appendChild(childOFChild);

            n.appendChild(child);

            Node guardNode = getNodeChild(c.xmlnode, "cond");

            if(guardNode !=null){
                String guard= "";
                guardNode = getNodeChild(guardNode,"text");

                if(guardNode.hasChildNodes()){
                    NodeList namelist  = guardNode.getChildNodes();
                    for(int i=0;i<namelist.getLength(); i++){
                         guard = namelist.item(i).getNodeValue();
                    }
                }

                child = document.createElement("cond");
                child.setAttribute("id","id"+lopeid++);

                childOFChild = document.createElement("posattr");
                childOFChild.setAttribute("x", Integer.toString(pX));
                childOFChild.setAttribute("y", Integer.toString(pY+10));
                child.appendChild(childOFChild);

                childOFChild = document.createElement("text");
                t = document.createTextNode(guard);
                childOFChild.appendChild(t);
                child.appendChild(childOFChild);
                n.appendChild(child);
            }

            node = getNodeChild(c.xmlnode, "time");

            if(node != null){
                child = document.createElement("time");
                child.setAttribute("id", "id"+lopeid++);

                childOFChild = document.createElement("posattr");
                childOFChild.setAttribute("x", Integer.toString(pX));
                childOFChild.setAttribute("y", Integer.toString(pY-5));
                child.appendChild(childOFChild);
                String transTime = "";

                if(node.hasChildNodes()){
```

```
                    NodeList namelist  = node.getChildNodes();
                    for(int i=0;i<namelist.getLength(); i++){
                        transTime = namelist.item(i).getNodeValue();

                    }
                }

                childOFChild = document.createElement("text");
                t = document.createTextNode(transTime);
                childOFChild.appendChild(t);
                child.appendChild(childOFChild);
                n.appendChild(child);
            }

        }

        /* Generate necessary XML codes for an arc */
        if(c instanceof Arc){
            Arc a = (Arc) c;

            // We'll try w/o this
            if(getNodeChild(c.xmlnode,"seg-conn")!=null){
                child = document.createElement("seg-conn");
                child.setAttribute("curv",
                                   getNodeAttribute(getNodeChild(c.xmlnode,
                                                                 "seg-conn"),
                                                    "curv"));
                n.appendChild(child);
            }

            child = document.createElement("placeend");
            child.setAttribute("idref", "id"+Integer.toString(a.xmlPlaceend));
            n.appendChild(child);

            child = document.createElement("transend");
            child.setAttribute("idref", "id"+Integer.toString(a.xmlTransend));
            n.appendChild(child);

            Node node = getNodeChild(c.xmlnode, "annot");

            if(node !=null){
                 Node posNode = getNodeChild(node, "posattr");
                 if(posNode != null){
                     try{
                         pX = rn.xmlX + (int) Double.parseDouble(
                             getNodeAttribute(posNode, "x")) - rn.relX;
                         pY = rn.xmlY + (int) Double.parseDouble(
                             getNodeAttribute(posNode, "y")) - rn.relY;
                     }
                     catch(NumberFormatException ex){
                         ex.printStackTrace();
                         pX = rn.xmlX - rn.relX;
                         pY = rn.xmlY - rn.relY;
                     }
```

```
                 }

                String arcInscr = "";
                node = getNodeChild(node,"text");
                if(node !=null && node.hasChildNodes()){
                    NodeList namelist  = node.getChildNodes();
                    for(int i=0;i<namelist.getLength(); i++){
                        arcInscr = namelist.item(i).getNodeValue();
                    }
                }

                child = document.createElement("annot");
                child.setAttribute("id", "id"+lopeid++);

                Element childOFChild = document.createElement("text");
                Text t = document.createTextNode(arcInscr);
                childOFChild.appendChild(t);
                child.appendChild(childOFChild);

                childOFChild = document.createElement("posattr");
                childOFChild.setAttribute("x", Integer.toString(pX));
                childOFChild.setAttribute("y", Integer.toString(pY));
                child.appendChild(childOFChild);

                n.appendChild(child);

            }
        }
        ele.appendChild(n);
    }

    /**
     * Save the specification as XML file
     */
    protected void saveProject (String filename, JPanel panel,
                                JPanel templatepanel) {
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance ();
        try {
            DocumentBuilder builder = factory.newDocumentBuilder ();
            Document doc = builder.newDocument ();
            Element t = null, r = null; // templates and rules
            Element rws = doc.createElement ("rws");
            Element p = doc.createElement("workspace");

            /* Save the templates*/
            if (templatepanel != null) {
                t = doc.createElement ("templates");
                for (int i = 0; i < templatepanel.getComponentCount (); i++) {
                    if (templatepanel.getComponent (i) instanceof JPanel &&
                        templatepanel.getComponentCount () >= 1) {
                        t.appendChild (
                            RWSNode.createElement (
                                doc, (RWSNode) ((JPanel)
                                templatepanel.getComponent (i)).
```

```
                                getComponent (0)));
                    }
                }
            }
            /* Save the RWSNodes */
            for (int i = 0; i < panel.getComponentCount (); i++) {
                p.appendChild (RWSNode.createElement (doc, (RWSNode)
                                                      panel.getComponent (i)));
            }
            /* Save rules */
            r = RWSConnector.createRulesElement (doc);

            if (t != null)
                rws.appendChild (t);
            rws.appendChild (p);
            if (r != null)
                rws.appendChild (r);
            doc.appendChild (rws);
            DOMSource source = new DOMSource (doc);
            FileOutputStream out = new FileOutputStream (filename);
            StreamResult result = new StreamResult (out);

            TransformerFactory tFactory =
                TransformerFactory.newInstance ();
            Transformer transformer = tFactory.newTransformer();
            Properties prop = new Properties ();
            prop.setProperty (OutputKeys.METHOD, "xml");
            prop.setProperty (OutputKeys.INDENT, "yes");
            prop.setProperty (OutputKeys.ENCODING, "ISO-8859-1");
            transformer.setOutputProperties (prop);
            transformer.transform(source, result);
            out.close ();
        }
        catch (Exception e) {
            e.printStackTrace ();
        }
    }

    /**
     * This method loads the specification from file and
     * uses the buildNode method to build the component objekts.
     */
    protected void openProject (String filename, JPanel templatePanel,
                                JPanel panel, RWSEditorFrame frame) {
        File file = new File (filename);
        connectorsToConnect = new HashMap ();
        allConnectors = new HashMap ();
        templateNodes = new HashMap ();

        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance ();
        try{
            DocumentBuilder builder = factory.newDocumentBuilder ();
            Document doc = builder.parse (file);

```

```
            Element rws = doc.getDocumentElement ();
            NodeList templates = ((Element)
                                  rws.getElementsByTagName ("templates").
                                  item (0)).getElementsByTagName ("node");
            NodeList workspace = ((Element)
                                  rws.getElementsByTagName ("workspace").
                                  item (0)).getElementsByTagName ("node");
            NodeList rules = ((Element)
                              rws.getElementsByTagName ("rules").
                              item (0)).getElementsByTagName ("rule");

            JLabel templatelabel = new JLabel("Components:");
            templatePanel.add(templatelabel);

            /* Read templates */
            for (int i = 0; i < templates.getLength (); i++) {
                RWSNode node = buildNode ((Element) templates.item (i));
                JPanel np = new JPanel(null);
                np.setPreferredSize (new Dimension(node.getNodeLength () +
                                                   node.borderWidth () * 2,
                                                   node.getNodeLength () +
                                                   node.borderWidth () * 2));

                np.addMouseMotionListener (frame);
                np.add (node);
                np.validate ();
                np.repaint ();
                templatePanel.add (np);
            }
            /* Read the specification */
            for (int i = 0; i < workspace.getLength (); i++) {
                RWSNode node = buildNode ((Element) workspace.item (i));
                panel.add (node);
            }
            /* Read rules */
            HashMap level1 = new HashMap ();
            HashMap level2;
            for (int i = 0; i < rules.getLength (); i++) {
                Element rule = (Element) rules.item (i);
                Integer from = new Integer (rule.getAttribute ("from"));
                Integer to = new Integer (rule.getAttribute ("to"));
                if (level1.containsKey (from))
                    level2 = (HashMap) level1.get (from);
                else {
                    level2 = new HashMap ();
                    level1.put (from, level2);
                }
                level2.put (to, to);
            }
            if (! level1.isEmpty ())
                RWSConnector.setRules (level1);

            templatePanel.validate ();
            templatePanel.repaint ();
```

```
            panel.validate ();
            panel.repaint ();

            /* Connect connectors to their respective neighbours */
            Iterator it = allConnectors.values ().iterator ();
            while (it.hasNext()) {
                RWSConnector c = (RWSConnector) it.next ();
                if (connectorsToConnect.containsKey (c)) {
                    String key = (String) connectorsToConnect.get (c);
                    RWSConnector peer =
                        (RWSConnector) allConnectors.get (key);
                    c.neighbour = peer;
                    c.unsetActive ();
                }
            }

            /* Set the counter */
            RWSNode.setCounter (maxId + 1);
        }
        catch (Exception e) {
            e.printStackTrace ();
        }
    }

    /*
     * This method builds the components.
     */
    private RWSNode buildNode (Element nodeElement) {
        RWSNode node = new RWSNode ();

        node.setId (Integer.parseInt (
                        nodeElement.getAttribute ("id")));

        maxId = (node.getId () > maxId) ? node.getId () : maxId;

        if (nodeElement.hasAttribute ("templref")) {
            RWSNode tpl = (RWSNode) templateNodes.get (
                new Integer (nodeElement.getAttribute ("templref")));
            node.setTemplate (tpl);
            node.isTemplate = false;
        }
        else {
            templateNodes.put (new Integer (node.getId ()), node);
            node.isTemplate = true;
        }

        /* Read the info element */
        Element info =
            (Element) nodeElement.getElementsByTagName ("info").
            item (0);
        if (info.hasAttribute ("componenttype"))
            node.setComponentType (
                info.getAttribute ("componenttype"));
        node.setNodeLength (Integer.parseInt (
```

```
                            info.getAttribute ("nodelength")));
        node.setStatus (Integer.parseInt (
                        info.getAttribute ("status")));

        /* Read the placement element */
        Element placement = (Element)
            nodeElement.getElementsByTagName ("placement").item (0);
        node.setExternalCoordinates (
            new Rectangle (
                Integer.parseInt (placement.getAttribute ("x")),
                Integer.parseInt (placement.getAttribute ("y")),
                Integer.parseInt (placement.getAttribute ("width")),
                Integer.parseInt (placement.getAttribute ("height"))));
        node.setCenterX (Integer.parseInt (
                         placement.getAttribute ("centerX")));
        node.setCenterY (Integer.parseInt (
                         placement.getAttribute ("centerY")));

        /* If only one connector, read the endcoordinates element */
        if (nodeElement.getElementsByTagName ("endcoordinates") !=
            null &&
            nodeElement.getElementsByTagName ("endcoordinates").
            getLength () > 0) {
            Element endcoordinates = (Element)
                nodeElement.getElementsByTagName (
                    "endcoordinates").item (0);
            node.setEndP1X (Integer.parseInt (
                            endcoordinates.
                            getAttribute ("endp1x")));
            node.setEndP1Y (Integer.parseInt (
                            endcoordinates.
                            getAttribute ("endp1y")));
            node.setEndP2X (Integer.parseInt (
                            endcoordinates.
                            getAttribute ("endp2x")));
            node.setEndP2Y (Integer.parseInt (
                            endcoordinates.
                            getAttribute ("endp2y")));
        }

        /* Connectors */
        NodeList clist = nodeElement.
            getElementsByTagName ("connector");
        RWSConnector [] connectors =
            new RWSConnector [clist.getLength ()];
        for (int j = 0; j < clist.getLength (); j++) {
            connectors [j] = new RWSConnector ();
            connectors [j].setNode (node);

            Element conn = (Element) clist.item (j);
            connectors [j].setIndex (Integer.parseInt (
                                     conn.getAttribute ("index")));
            connectors [j].isTemplate =
                (conn.getAttribute ("istemplate").equals("true")) ?
```

```
                true : false;

            Element pos =
                (Element) conn.getElementsByTagName ("pos").
                item (0);
            connectors [j].setP (
                new Point (
                    Integer.parseInt (pos.getAttribute ("x")),
                    Integer.parseInt (pos.getAttribute ("y"))));

            NodeList neighbours = conn.getElementsByTagName ("neighbour");
            if (neighbours != null && neighbours.getLength () > 0) {
                Element neighbour = (Element) neighbours.item (0);
                String val = neighbour.getAttribute ("node") + ":" +
                    neighbour.getAttribute ("index");
                connectorsToConnect.put (connectors [j], val);
            }

            Element conninfo =
                (Element) conn.getElementsByTagName ("info").
                item (0);
            connectors [j].setStatus (
                Integer.parseInt (conninfo.getAttribute ("status")));
            connectors [j].setConnectorType (
                Integer.parseInt (conninfo.getAttribute ("connectortype")));

            String key = connectors [j].node.getId () + ":" +
                connectors [j].getIndex ();
            allConnectors.put (key, connectors [j]);
            connectors [j].validate ();
        }

        node.setBounds(node.getExternalCoordinates ());
        node.setConnectors (connectors);
        int [] connectorX = new int [connectors.length];
        int [] connectorY = new int [connectors.length];
        int x = node.getExternalCoordinates ().x;
        int y = node.getExternalCoordinates ().y;

        for (int i = 0; i < connectors.length; i++) {
            connectorX [i] = x + connectors [i].getP ().x;
            connectorY [i] = y + connectors [i].getP ().y;
        }

        node.setConnectorX (connectorX);
        node.setConnectorY (connectorY);
        node.addConnectors ();
        node.validate ();
        return node;
    }
}
```

Listing 10: BackgroundPanel.java

```
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
import java.util.HashMap;
import java.util.Vector;
import java.util.Enumeration;
import java.awt.Graphics2D;
import java.awt.BasicStroke;

import java.io.*;

/**
 * The panel on which the specification is drawn.
 */
class BackgroundPanel extends JPanel implements Scrollable,
                                                MouseMotionListener {
    private int maxUnitIncrement = 1;

    Dimension size = new Dimension (900, 800);
    RWSEditorFrame parent;

    BackgroundPanel () {
        setAutoscrolls (true);
        addMouseMotionListener (this);
    }

    BackgroundPanel (RWSEditorFrame parent) {
        this.parent = parent;
        setLayout (null);
    }

    /**
     * Interface method
     */
    public void mouseMoved (MouseEvent e) {}

    /**
     * Interface method
     */
    public void mouseDragged (MouseEvent e) {
        /* The user is dragging us, so scroll! */
        Rectangle r = new Rectangle (e.getX(), e.getY(), 1, 1);
        scrollRectToVisible (r);
        this.setAutoscrolls (true);
    }

    public void setMaxUnitIncrement (int pixels) {
        maxUnitIncrement = pixels;
    }

    /**
     * Interface method
     */
    public boolean getScrollableTracksViewportHeight () {
```

```
        return false;
    }

    /**
     * Interface method
     */
    public boolean getScrollableTracksViewportWidth () {
        return false;
    }

    /**
     * Interface method
     */
    public Dimension getPreferredScrollableViewportSize () {
        return new Dimension (800, 600);
    }

    /**
     * Interface method
     */
    public int getScrollableBlockIncrement (Rectangle visibleRect,
                                            int orientation,
                                            int direction) {
        if (orientation == SwingConstants.HORIZONTAL)
            return visibleRect.width - maxUnitIncrement;
        else
            return visibleRect.height - maxUnitIncrement;
    }

    /**
     * Interface method
     */
    public int getScrollableUnitIncrement (Rectangle visibleRect,
                                           int orientation,
                                           int direction) {
        /* Get the current position. */
        int currentPosition = 0;
        if (orientation == SwingConstants.HORIZONTAL) {
            currentPosition = visibleRect.x;
        } else {
            currentPosition = visibleRect.y;
        }

        /**
         * Return the number of pixels between currentPosition and the
         * nearest tick mark in the indicated direction.
         */
        if (direction < 0) {
            int newPosition = currentPosition -
                (currentPosition / maxUnitIncrement)
                * maxUnitIncrement;
            return (newPosition == 0) ? maxUnitIncrement : newPosition;
        } else {
            return ((currentPosition / maxUnitIncrement) + 1)
```

```
                    * maxUnitIncrement
                    - currentPosition;
            }
        }

        public Dimension getPreferredSize () {
            return size;
        }

        /**
         * Paint this component
         */
        public void paintComponent (Graphics g) {
            super.paintComponent(g);

            g.setColor(Color.white);
            g.fillRect(0, 0, size.width, size.height);
            if(RWSEditorFrame.justTheLine){
                RWSEditorFrame.justTheLine = false;
                Graphics2D g2 = (Graphics2D) g;
                float [] dash = new float [] {5};
                g2.setStroke(new BasicStroke(1,
                                             BasicStroke.CAP_ROUND,
                                             BasicStroke.JOIN_ROUND,
                                             (float) 1,
                                             dash,
                                             (float) 1));
                g2.setColor(Color.black);
                if ( RWSConnector.selectedConnector != null &&
                     RWSConnector.selectedConnector.isActive() ){
                    g2.drawLine(RWSConnector.selectedConnector.externalCenterX(),
                                RWSConnector.selectedConnector.externalCenterY(),
                                RWSEditorFrame.mouseX,
                                RWSEditorFrame.mouseY);

                }
                else
                    g2.drawLine(parent.startX,
                                parent.startY,
                                RWSEditorFrame.mouseX,
                                RWSEditorFrame.mouseY);
            }
        }
}
```

Listing 11: ConnectConnectorToCPNNodeFrame.java

```
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
import java.util.HashMap;
import java.util.Vector;
import java.util.Enumeration;
import java.util.Iterator;
```

```java
import java.util.Vector;
import java.awt.Graphics2D;
import java.awt.BasicStroke;

/**
 * Class for opening a pop-up window to connect a RWSConnector
 * (corresponding to the interface nodes in the specification
 * language) to it's CPN counterpart
 */
class ConnectConnectorToCPNNodeFrame extends JFrame implements ActionListener {

    JComboBox comps, inters;
    JButton button;
    JLabel label1, label2;
    Container pane;
    RWSConnector connector;
    HashMap interfaces, cpnNodes;
    Vector interFaceList;

    ConnectConnectorToCPNNodeFrame ( RWSConnector connector ) {
        if (XMLUtils.cpnComponents.size () < 1){
            /* No cpn components added yet */
            dispose();
            return;
        }

        Iterator it = XMLUtils.cpnComponents.keySet ().iterator ();
        String [] components = new String [XMLUtils.cpnComponents.size ()];
        int i = 0;
        interfaces = new HashMap ();
        while (it.hasNext()) {
            components [i] = (String) it.next ();
            cpnNodes = (HashMap) XMLUtils.cpnComponents.get (components [i]);
            Iterator it2 = cpnNodes.values ().iterator ();
            interFaceList = new Vector ();
            while (it2.hasNext()) {
                CPNNode nd = (CPNNode) it2.next ();
                if (nd instanceof Place) {
                    interFaceList.add (nd.getId());
                }
            }
            interfaces.put (components [i], interFaceList);
            i++;
        }

        this.connector = connector;
        setDefaultLookAndFeelDecorated (true);
        setDefaultCloseOperation (DISPOSE_ON_CLOSE);
        pane = getContentPane ();
        label1 = new JLabel ("Component␣name:␣");
        label2 = new JLabel ("Interface␣id:␣");
        comps = new JComboBox (components);
        comps.setActionCommand ("comps");
        inters = new JComboBox ((Vector) interfaces.get (components [0]));
```

```
        comps.addActionListener (this);
        button = new JButton ("Click!");
        JPanel bg = new JPanel ();
        bg.add (label1);
        bg.add (comps);
        bg.add (label2);
        bg.add (inters);
        bg.add (button);
        pane.add (bg);
        button.addActionListener (this);
    }

    public void actionPerformed (ActionEvent e){
        if (e.getSource () instanceof JComboBox) {
            if (((JComboBox) e.getSource ()).getActionCommand ().
                equals("comps")) {
                Vector ifaces = (Vector)
                    interfaces.get (((JComboBox) e.getSource ()).
                                    getSelectedItem ().toString ());
                inters.removeAllItems ();
                for (int i = 0; i < ifaces.size (); i++)
                    inters.addItem (ifaces.elementAt (i));
            }
        }
        else{
            connector.addCPNInterface (inters.getSelectedItem ().toString (),
                                       comps.getSelectedItem ().toString ());
            dispose ();
        }
    }
}
```

Listing 12: ChangeToolTipText.java

```
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
import java.util.HashMap;
import java.util.Vector;
import java.util.Enumeration;
import java.awt.Graphics2D;
import java.awt.BasicStroke;

/**
 * Small class to open a pop-up window for changing the descriptive
 * names of RWSNodes (atomic components).
 */
class ChangeToolTipText extends JFrame implements ActionListener {

    JTextField text;
    JButton button;
    JLabel label;
    Container pane;
    RWSNode parent;
```

```

    ChangeToolTipText (RWSNode parent) {
        this.parent = parent;

        if (parent == null)
            dispose ();

        setDefaultLookAndFeelDecorated (true);
        setDefaultCloseOperation (DISPOSE_ON_CLOSE);
        pane = getContentPane ();

        label = new JLabel ("Text: ");
        text = new JTextField (20);
        text.setText (parent.getToolTipText ());
        button = new JButton ("Click!");

        JPanel bg = new JPanel();
        bg.add (label);
        bg.add (text);
        bg.add (button);
        pane.add (bg);
        button.addActionListener (this);
        pack ();
        setVisible (true);
    }

    public void actionPerformed (ActionEvent e) {
        parent.setToolTipText (text.getText ());
        dispose ();
    }
}
```

Listing 13: CreateMultipleNodesFrame.java

```
import java.awt.event.*;
import java.awt.*;
import javax.swing.*;
import java.util.HashMap;
import java.util.Vector;
import java.util.Enumeration;
import java.awt.Graphics2D;
import java.awt.BasicStroke;

/**
* The class pops up a window so that the user may specify an integer.
* This is the number of components created when generating multiple
* connected components to facilitate fast construction. (Used on
* components with two connectors).
*/
class CreateMultipleNodesFrame extends JFrame implements ActionListener {

    JTextField text;
    JButton button;
    JLabel label;
```

```
    Container pane;
    RWSEditorFrame parent;

    CreateMultipleNodesFrame ( RWSEditorFrame parent ) {
        this.parent = parent;

        setDefaultLookAndFeelDecorated(true);
        setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        pane = getContentPane();

        label = new JLabel("Number␣of␣nodes:␣");
        text = new JTextField(4);
        button = new JButton("Click!");

        JPanel bg = new JPanel();
        bg.add(label);
        bg.add(text);
        bg.add(button);
        pane.add(bg);
        button.addActionListener(this);
        pack();
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e){
        try {
            parent.rwsConnectMultiple ( Integer.parseInt( text.getText () ) );
        }
        catch(NumberFormatException ex){
            ex.printStackTrace();
        }
        dispose();
    }
}
```

Listing 14: RWSFileFilter.java

```
import java.io.File;
import javax.swing.*;
import javax.swing.filechooser.*;

/**
 * File filter for opening and saving RWS files (specification) files
 */
public class RWSFileFilter extends FileFilter {

    String filter, description;

    public RWSFileFilter () {
        this.filter = "xml";
        this.description = "RWSEditor␣files";
    }

    public RWSFileFilter (String filter) {
```

```
        this.filter = filter.toLowerCase ();
        this.description = "RWSEditor␣files";
    }

    public RWSFileFilter (String filter, String description) {
        this.filter = filter.toLowerCase ();
        this.description = description;
    }

    /* Accept all directories and all rws files */
    public boolean accept (File f) {
        if (f.isDirectory ())
            return true;

        String extension = f.getName ().
            substring (f.getName ().lastIndexOf ('.') + 1).toLowerCase ();
        if (extension != null &&
            (extension.equals (filter)))
            return true;
        return false;
    }

    /* The description of this filter */
    public String getDescription () {
        return description;
    }
}
```

Listing 15: Resize.java

```
import java.awt.event.*;
import java.awt.*;
import java.lang.*;
import javax.swing.*;
import java.util.HashMap;
import java.util.Vector;
import java.util.Enumeration;
import java.awt.Graphics2D;
import java.awt.BasicStroke;

/**
 * Resize the size of the working space
 */
class Resize extends JFrame implements ActionListener {

    JTextField textX, textY;
    JButton button;
    JLabel labelX, labelY;
    Container pane;
    RWSEditorFrame parent;
    int columnsize = 10;
    String width;
    String height;
    Resize (RWSEditorFrame parent) {
```

```
        this.parent = parent;
        setDefaultLookAndFeelDecorated (true);
        setDefaultCloseOperation (DISPOSE_ON_CLOSE);
        pane = getContentPane ();
        width = Integer.toString (parent.panel.getWidth ());
        height = Integer.toString (parent.panel.getHeight ());

        labelX = new JLabel ("X: ");
        textX = new JTextField (width, columnsize);
        labelY = new JLabel ("Y: ");
        textY = new JTextField (height, columnsize);

        button = new JButton ("Click!");

        JPanel bg = new JPanel ();
        bg.add (labelX);
        bg.add (textX);
        bg.add (labelY);
        bg.add (textY);
        bg.add (button);
        pane.add (bg);
        button.addActionListener (this);
        pack ();
        setVisible (true);
    }


    public void actionPerformed (ActionEvent e) {
        try {
            if (textX.getText ().equals("") && textY.getText ().equals(""))
                parent.rwsEditorResize (Integer.parseInt (width),
                                        Integer.parseInt (height));
            else if (textX.getText ().equals(""))
                parent.rwsEditorResize (Integer.parseInt (width),
                                        Integer.parseInt (textY.getText ()));
            else if (textY.getText ().equals(""))
                parent.rwsEditorResize (Integer.parseInt (textX.getText ()),
                                        Integer.parseInt (height));
            else
                parent.rwsEditorResize (Integer.parseInt (textX.getText ()),
                                        Integer.parseInt (textY.getText ()));
        }
        catch (NumberFormatException ex) {
            ex.printStackTrace ();
        }
        dispose ();
    }
}
```

Listing 16: XMLFileFilter.java

```
import java.io.File;
import javax.swing.*;
import javax.swing.filechooser.*;
```

```java

/**
 * File filter for opening and saving xml files
 */
public class XMLFileFilter extends FileFilter {

    /* Accept all directories and all rws files */
    public boolean accept(File f) {
        if (f.isDirectory())
            return true;

        String extension = f.getName().
            substring (f.getName ().lastIndexOf ('.') + 1).toLowerCase ();
        if (extension != null &&
            (extension.equals ("xml")))
            return true;
        return false;
    }

    /* The description of this filter */
    public String getDescription() {
        return "Design␣/␣CPN␣XML␣export␣files";
    }
}
```