MASTER OF SCIENCE THESIS

A Domain-Specific Language for Tramway Control Systems

Rasmus Dyhrberg
s973403

Nikolaj Christensen
s973588

May 2004

Computer Science and Engineering
Informatics and Mathematical Modelling
Technical University of Denmark

Preface

This master thesis has been developed in the period from october 14th 2003 to May 21th 2004 at the department of Computer Science and Engineering at the Technical University of Denmark (DTU). It concludes our M.Sc. studies at DTU.

The project was supervised by Associate Professor, Ph.D. Anne E. Haxthausen.

Acknowledgements

We would like to thank our supervisor Anne E. Haxthausen for constructive criticism and inspiration throughout the project period.

Additionally, Professor Jan Peleska of University at Bremen is thanked for his help in answering questions.

We would also like to thank the following people for discussions and comments on the report and the models: Torben Gjaldbæk, Søren Dyhrberg, Ulla Dyhrberg, and Linda Nørskov.

Lyngby, May 21, 2004

Nikolaj Kaare Christensen   Rasmus Dyhrberg

In dedication to my dear departed brother Thomas Kaare Christensen.

Abstract

Domain specific languages can be used in the construction of computerised systems within a specific domain. In this thesis a model of a language for the domain of tramway networks in the former DDR is constructed. The language consists of a graphical representation of networks and a tables part containing data of routes in the networks.

The language facilitates the construction of a generic control system for tramway networks within the considered domain. A generic control system makes it possible to construct a control system for an arbitrary tramway network within the considered domain. The construction is based on a description of a concrete network formulated in the domain specific language, thereby, saving a substantial amount of time when designing a control system for a specific network.

A formal model of the domain specific tramway language (DST-L) is constructed, and the generic control components of the control system are designed and a formal model is constructed. Furthermore, how to interpret the control components to gain a running system is identified. When initialised with data from a description in the DST-L the generic control components constitute a control system for the specific network.

Since the safety is of high importance, as the task of tramway control systems is to ensure safety in a tramway network, safety predicates are formulated and the concept of how to prove that any network controlled by a control system, constructed using this generic method, fulfills the safety requirements is developed.

The formal models are refined according to the RAISE method to gain concrete models, and imperative models are developed to facilitate implementation of a prototype.

A prototype of the domain specific language and of the generic control system is implemented based of the imperative formal models. Furthermore, to demonstrate the functionality of the control system a simulator is implemented. Both the prototype and the simulator are implemented using XML technology and the imperative language C. Thus, in this thesis it is presented how to translate RSL to XML and RSL to C.

Keywords: Domain specific languages, formal methods, RAISE, C, XML, tramway control systems, safety, verification.

Resumé

Domæne specifikke sprog er behjælpelige ved konstruktion af computeriserede systemer indenfor et konkret domæne. I dette eksamensprojekt konstrueres en model af et sprog for domænet af sporvognsanlæg i det tidligere DDR. Sproget består af en grafisk fremstilling af netværk og en tabeldel indeholdende data om ruter i anlæg.

Sproget muliggør konstruktion af et generisk styresystem for sporvognsanlæg indenfor det betragtede domæne. Et generisk styresystem gør det muligt at konstruere et styresystem for et arbitrært sporvognsanlæg i domænet. Konstruktionen er baseret på en beskrivelse af et konkret anlæg formuleret i det domæne specifikke sprog. Dermed spares en ikke uvæsentlig mængde tid ved design af et styresystem for det specifikke anlæg.

En formel model af det domæne-specifikke-sporvognsanlægs-sprog (DST-L[1]) konstrueres, og de generiske styrekomponenter i styresystemet designes og en formel model af dem konstrueres. Ydermere identificeres det hvordan styrekomponenter kan fortolkes for at opnå et funktionsdygtigt styresystem. Når de er initialiseret med data fra en beskrivelse i det DST-L udgør styrekomponenterne et styresystem for det specifikke netværk.

Da sikkerheden er af særlig vigtighed, fordi et styresystems opgave er at sikre sikkerhed i et sporvognsanlæg, formuleres sikkerhedspredikater. Ydermere bliver det udviklet hvordan det kan bevises at alle anlæg styret af et styresystem, konstrueret med denne generiske metode, opfylder disse sikkerhedskrav.

De formelle modeller bliver forfinet i henhold til RAISE metoden for at opnå konkrete modeller, og imperative modeller bliver udviklet for at kunne implementere en prototype.

Baseret på de imperative formelle modeller implementeres en prototype af det domæne specifikke sprog samt af styresystemet. For at demonstrere funktionaliteten af styresystemet implementeres ydermere en simulator. Både prototypen og simulatoren bliver implementeret i XML og det imperative sprog C. Derfor præsenteres det i eksamensprojektet hvordan man kan oversætte RSL til XML og RSL til C.

Nøgleord: Domæne specifikke sprog, formelle metoder, RAISE, C, XML, sporvognsstyresystemer, sikkerhed, verifikation.

[1]Engelsk forkortelse.

Contents

List of Figures

Chapter 1

Introduction

This thesis deals with the topic of how a domain specific language can be used to construct a generic control system for tramway networks.

Domain specific languages (DSL) are developed to be able to describe an instance within a specific application domain. The strength of DSLs is the adoption of terms and concepts from the application domain to facilitate the communication between domain experts and the specialists developing computerised systems within the domain.

Wide spectrum languages such as UML use language elements which are independent of the context of the application domain. Unfortunately, the resulting specifications are often difficult to understand for people outside the computer science world. It is therefore questionable whether wide spectrum languages will ever serve as an useful means of communication between system engineers who are experts in the application domain and computer science specialists handling the software-related control aspects of a system. On the other hand, the advantage of wide spectrum languages is that a large variety of systems may be specified without changing the formalism.

Working with safety critical systems, a DSL is preferable. It reduces the risk of misunderstandings by facilitating the communication between domain experts and computer science specialists.

To be able to argue about specification correctness of a description formulated in a DSL unambiguously, the semantics of the language must be established in a rigorous way. As a consequence, the effort of developing a DSL is not trivial. Therefore, the necessary time is only invested in cases where the application domain is well understood and the criticality of applications implies that reasoning about specifications and implementations is mandatory.

The task of tramway control systems is to ensure safe operations of the tramway traffic to avoid accidents like tram collisions and derailments. Hence, the correctness of such control systems is highly critical. The domain of tramway networks is well understood by domain experts and therefore it is obvious to develop a DSL for tramways. To justify the time used to develop a domain specific tramway language (DST-L) the control system is made generic, facilitating construction of control systems for an arbitrary tramway network within the considered domain.

The process of constructing a control system is explained in the following (the process is illustrated in figure 1.1). A description of a specific tramway network is formulated in DST-L. Based on the description, configuration data is generated using an application developed for this specific purpose. The generated configuration data and a module of generic code constitute a control system for the specific tramway network.

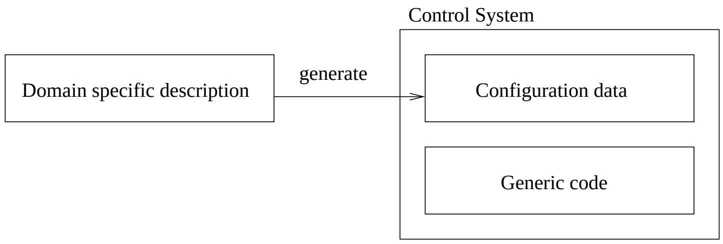

Figure 1.1: The process of constructing a control system.

The main objectives of this thesis are to develop a formal model of a DST-L for tramway networks which are used in the former DDR[1], a formal model of a generator to generate configuration data, and a formal model of the generic control system. These models constitute the base for formally stating the safety requirements of the network and how to verify that the control system ensures safety. Another important objective is an implementation of a prototype of the DST-L, the generator, and the control system.

1.1 Background

For some years a research project concerning tramway networks has been in progress. Associate Professor Anne E. Haxthausen, Technical University of Denmark, and Professor Jan Peleska, University of Bremen, have in cooperation been researching on the construction of a DST-L. A number of articles concerning the results of the research have been published.

In 2002 the article "A Domain Specific Language for Railway Control Systems" [HP02] was published, describing the development of a DSL for railway control systems[2].

In 2003 the next article "Generation of Executable Railway Components from Domain specific Descriptions" [HP03b] was published. It concerns the topic of how to generate control systems from descriptions written in the DSL described in the first article.

Later the same year a third article was published. The title was "Automatic Verification and Test for Railway Control Systems Based on Domain-Specific Descriptions" [HP03a], and it presented a method for the verification of route based railway control systems developed from the domain specific descriptions.

Concerning this thesis, the most important topics in the articles are as follows:

The engineering concepts of a DST-L have been formulated in [HP02] and are used with slight modifications. It is presented by a network description, given by a graphical representation of a concrete network within the considered domain, and a number of interlocking tables. The purpose of the interlocking tables is to fix the possible routes through the network and the safety conditions which are sufficient to allow trams to safely pass along these routes.

The safety requirements have been formulated in [HP03a] and a suggestion for a safety monitor is presented in [HP02].

[1]Possibly it also fits other networks.
[2]Or tramway control systems.

The idea of using state transition system (STS) theory for the generic part of the control system and for the physical environment has been presented in [HP03b], [HP03a] and an STS semantics suggestion is presented in [HP03a].

This thesis is an extension of the research project. Therefore, the results of the research project are collected and adapted to form the foundation of the thesis.

1.2 Thesis Objectives

The main objectives of this thesis are presented earlier in this chapter. Here they are summarised:

1. To develop a formal model of the DST-L.
2. To develop a formal model of a generic control system.
3. To develop a formal model of a generator of configuration data for the generic control system.
4. To state the safety requirements and formulate the proof obligations for the control system ensuring safety.
5. To implement a prototype of the DST-L, the generator and the control system.

1.3 Approach

The specification of the formal models is developed using the formal specification language RSL [Gro92].

The approach has been to:

1. Establish an initial model of the DST-L, the generator and the control system.
2. Express the safety requirements which the control system must fulfill in terms of the model.
3. Informally argue pro the initial model satisfying the safety requirements.
4. Develop imperative versions of the models.
5. Implement the models of the DST-L, the generator, and the control system.
6. Implement a simulator of the functionality of the generic control system.

1.4 Thesis Outline

In this thesis the theory and discussion parts are provided ”on a need to know basis” - meaning that theory is introduced as needed and some choices are discussed as they are presented. A chapter is provided towards the end of the thesis summarising the discussion and providing discussion of additional topics. This structure has been chosen to give a smoother reading of the report, trying to minimise the need for referring to other chapters than the one being read.

Additionally, to make the thesis more readable each chapter has an introduction and in general a chapter summary.

The thesis outline is:

- Chapter 2 presents an informal description of the tramway domain.

- Chapter 3 presents in details the engineering concepts that are the foundation of this thesis.
- Chapter 4 concerns method and model overview.
- Chapters 5-8 describe the RSL models. Please note that chapter 6 does not actually describe an RSL model but forms the basis for the creation of the models presented in the subsequent chapters.
- In chapter 9 the safety requirements for a tramway network are stated informally and modelled in RSL.
- Chapter 10 presents the procedure used for verifying the safety along with the RSL proof obligations.
- The models are further developed with the intention of implementation in chapter 11.
- A control algorithm for the control system is presented in chapter 12.
- Chapters 13 - 15 present the implementation of the different models.
- Chapter 16 concerns the methods used for testing the different parts of the implementation.
- A simulation of the running control system for a specific network is presented in chapter 17.
- The different aspects of models, implementation, and verification are discussed in chapter 18.
- Chapter 19 presents possible extensions to the thesis and future work.
- Chapter 20 presents conclusions on the work of the thesis.

The appendices contain the different models, programs, and etc. as follows:

- Appendix A contains a basic description of the syntax of STSs.
- Appendix B contains the different RSL models.
- Appendix C contains the proof obligations for ensuring the safety of the tramway network.
- Appendix D contains an informal argumentation (verification) of the proof obligations, thereby, informally arguing that the tramway network is safe.
- Appendix E contains the developed imperative models.
- Appendix F contains the control algorithm.
- Appendix G contains the implementation of the DST-L in XML and XSLT.
- Appendix H contains the implementation of the STS-L in C.
- Appendix I contains the implementation of the generator in XSLT.
- Appendix J contains instantiation of two example networks, called the running example network and second network. The instantiation consists of description in the DST-L in XML and initialized control components in C.
- Appendix K contains a program for simulating the running of the control system.
- Appendices L and M contain test programs for the XML part and the results, respectively.
- Appendices N and O contain test programs for the C programs and the results, respectively.
- Appendix P contains a user's guide to the GUI for collected data of the DST-L.

1.5 Prerequisites

To read and fully understand this thesis the prerequisites are:

- Basic understanding of computer science and formal methods.

- Knowledge of the RSL specification language [Gro92].
- Knowledge of the C programming language [KR88].
- Knowledge of the eXtensible Markup Language (XML) and related tools (XSLT, DTD) [HM02], [Dub03].

Chapter 2

Informal Description of the Tramway Domain

This chapter presents an informal description of tramways and related aspects.

The tramways are described as they are used in various cities in the former DDR.

2.1 Tramways

Tramways are composed of tramway networks, and trams, which are presented in this section.

2.1.1 Tramway Networks

Tramway networks are physical structures spanning over a certain geographical area. Networks consists of tracks which are joined in junctions and can cross in crossings.

Track A *track* consists of two parallel iron bars, called rails, on which trams can move.

Crossing A *crossing* is a construction enabling two tracks to cross if they are in the same horizontal level. An illustration of a crossing is given in figure 2.1. The possible ways to traverse a crossing are a to b, c to d and vice versa.

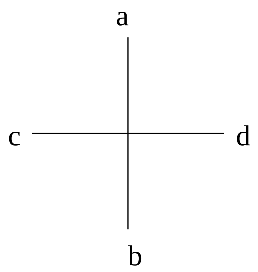

Figure 2.1: Crossing.

Junction A *junction* is a construction of joined tracks. A junction has a *stem*, a *turn branch*, and a *straight branch*. Furthermore, each junction has a point determining how trams can move between the stem and the branches. An illustration of a junction is given in figure 2.2.

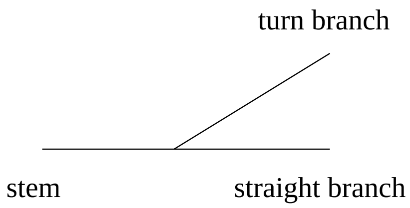

Figure 2.2: Junction.

Point A *point* is a connector of tracks in a junction. It is located where the stem, and the two branches meet. A point determines how the trams can move in the junction. A point can be either non controllable or controllable.

A *non controllable point* is a static connector of tracks, meaning that the paths through the junction are static. It is possible to move from the turn and straight branch to the stem and from the stem to the straight branch.

A *controllable point* has a pair of movable rails, called *tongues*. A controllable point may be in turn, straight, or intermediate position (i.e. switching from turn to straight or vice versa). The position of a point corresponds to the position of the two tongues. While switching from one position it is possible to reverse the direction of the switching. For instance, a controllable point switching from turn to straight can be set to shift back to turn. This is to prevent the point being stuck in intermediate position, for example if a stone or similar blocks it.

It is possible to move from the stem to the turn branch if the point is in turn position (figure 2.3), and from the stem to the straight branch if the point is in straight position (figure 2.4). Furthermore, it is possible to move from the turn branch to the stem if the point is in turn position, but also if the point is in straight position. In this situation the tram mechanically switches the point position. Similarly, it is possible to move from the straight branch to the stem whether the point is in turn or straight position. Note, this mechanical switching is only possible if the tram drives slowly through the point.

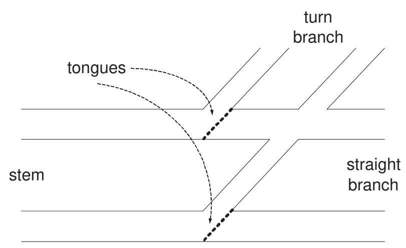

Figure 2.3: Controllable point in turn position.

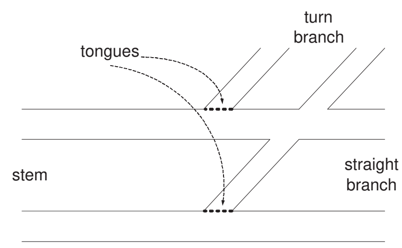

Figure 2.4: Controllable point in straight position.

Sub-Network A tramway network can be divided into *sub-networks*. Sub-networks are connected with linear tracks, at locations called borders. In figure 2.5 an illustration of two sub-networks is given[1]. The reason for this division is that city-networks consist of areas with heavy traffic, such as stations or multiple crossing rails, and areas with long linear rails. The station areas are controlled while the linear rails does not need to be controlled. Thus, the sub-network considered are stations or similar areas.

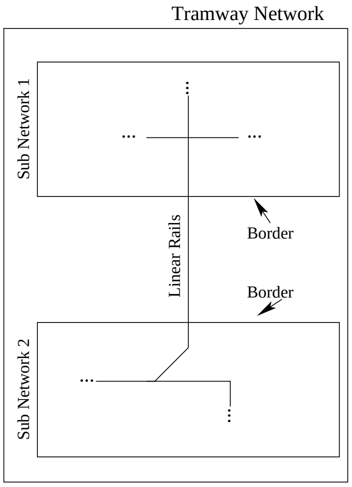

Figure 2.5: Connection of sub-networks.

2.1.2 Trams

A *tram* is a vehicle constructed to drive in tramway networks. Each tram is driven by a *tram driver*, deciding the speed and the direction of the tram.

Trams may have an *onboard control computer* enabling communication with a computer system controlling the network.

In a junction with a controllable point the point position determines the direction of trams. If the point position is intermediate the tram might be derailed.

2.2 Safety

Safe tramways always comply with the following two statements:

- No collisions involving trams.
- No derailments of trams.

2.3 Monitoring and Controlling Traffic

In a tramway network certain devices can be used for detection of trams and for controlling traffic.

[1]In the rest of this thesis only such sub-networks are considered. Therefore, the term "network", in the following sections and chapters, refers to such sub-networks.

2.3.1 Sensors

A device, called a *sensor*, is used for detection of trams in a tramway network. Sensors are motion sensors which register when a tram is passing by. A motion sensor is in one of two states; *high* or *low*. When a tram passes a sensor, the state cycles of the sensor is as follows;

- The sensor is initially *low*.
- When the tram moves in front of the sensor, it changes to *high*.
- The sensor remains *high* for a time period Δt.
- When the time period Δt has passed, the sensor returns to *low*. The time Δt is less than the time it takes for a tram to pass by the sensor. Therefore, a sensor may be in state *low* even though a tram is passing it.
- The sensor remains *low* until another tram moves in front of the sensor.

Figure 2.6 illustrates a tram passing a sensor and the changes of the sensor state.

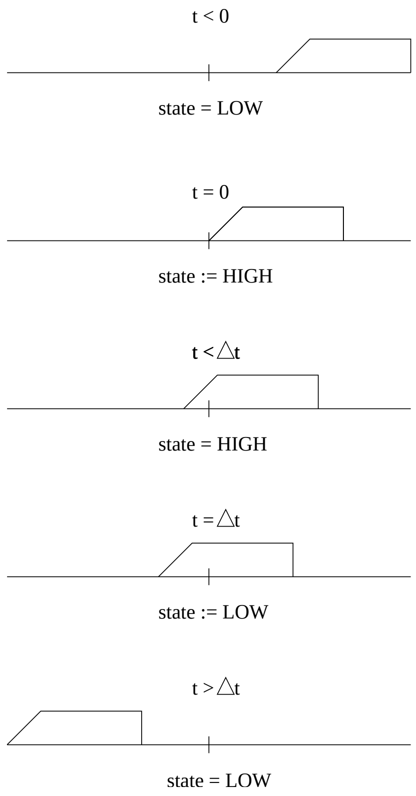

Figure 2.6: Tram passing sensor.

Sensors are placed at critical locations in a network. At the border of a network sensors, called *border sensors*, monitor traffic entering (*entry border sensor*) and leaving (*exit border sensor*) the network. Three sensors flank junctions, namely, on the stem, the turn branch and the straight branch. These sensors are called *stem sensor*, *turn sensor* and *straight sensor* respectively and are collectively called *point sensors*. For junctions with a controllable point, the distance from the point to the flanking sensors must be slightly longer than the length of a tram, in order to detect the actual direction of the tram immediately after it has passed the point. For non controllable points only the sensor at the stem must be located close to the point. The distance from the point to the sensors on the branches must be at least slightly more than the length of a tram, otherwise a single train can be located at more than one segment, which makes it impossible to determine specificly the location of the tram in the network.

2.3.2 Signals

Signals located at sensors are used to restrict the movement of trams in tramway networks. A signal is visible in one direction only. The tram driver is informed by the signal whether the tram is allowed to continue or not. Furthermore, the signal indicates which direction the tram drives after having passed the signal. Therefore, the following *signal aspects* are possible:

- *stop*: The tram should stop.
- *go straight*: The tram may proceed in straight direction.
- *go left*: The tram may proceed in left direction.
- *go right*: The tram may proceed in right direction.

The last two aspects are only relevant if the signal is located near the stem of a junction with a controllable point.

Signals are generally placed near the borders of networks to control incoming traffic.

2.4 Properties of Junctions

Junctions form the backbone of a network. Junctions makes it possible to connect rails and are therefore necessary to distribute traffic. On the other hand junctions also increase the likelyhood of derailment and collision. Thus, the properties of junctions with respect to how they should be used, and in which cases they must not be used are very important. Consequently, these properties are therefore described in this section.

2.4.1 Logical Direction

The possible movement in junctions is as described in section 2.1.1, and depends on whether the junction has a controllable point or a non controllable point. Nevertheless, it is meaningful to talk about a logical direction in junctions. The logical direction through a junction with a non controllable point is from the branches to the stem, and the logical direction in a controllable point is from the stem to the branches. This is because a non controllable point is static and can only merge tracks, and a controllable point is dynamic and can guide trams in different directions. Therefore, if a junction is only approached from the two branches, a non controllable point is sufficient, and if the junction is approached from the stem, a controllable point is needed.

2.4.2 Connection of Junctions

When introducing control of traffic it must be possible to uniquely describe a path through a network by a sensor sequence. The domain description so far allows constructions where this is not possible. This concerns connection of more than one junction. Therefore, it is necessary to formulate constraints on how junctions can be connected.

The first necessary constraint is as follows; The three sensors flanking a junction must be distinct. If for instance the turn sensor is the same as the straight sensor, it is not possible to decide the path from the stem sensor to the turn/straight sensor. The situation is illustrated in figure 2.7.

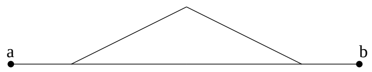

Figure 2.7: Construction of junctions with identical turn and straight branch sensor.

It is allowed for two junctions to have one sensor in common, except the stem sensor. Figure 2.8 illustrates two possible constructions.

Figure 2.8: Allowable constructions of two junctions with one sensor in common.

It is allowed for two junctions to have the two branching sensors in common. Figure 2.9 illustrates an allowed construction.

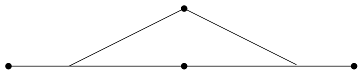

Figure 2.9: Allowable construction of two junctions with two sensors in common.

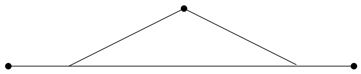

Figure 2.10: Not allowable construction of two junctions with three sensors in common.

Furthermore it has been decided not to allow two junctions to have three sensors in common. This means that a construction like the one in figure 2.10 is not allowed. This is mostly to simplify the possible network constructions.

2.5 Chapter Summary

In this chapter the domain of tramways has been described. It consists of a tramway network and trams. A network consists of tracks, junctions, crossings, and devices, being points, sensors, and signals. Properties of the different track constructions and devices has been presented.

Chapter 3

Engineering Concepts

This chapter presents the concepts involved in constructing a generic control system for tramway networks. The concepts were conceived by Anne E. Haxthausen at Technical University of Denmark and Jan Peleska at University of Bremen, and developed through a series of articles [HP02], [HP03b], and [HP03a].

Section 3.1 contains an overview of the overall approach. Section 3.2 introduces routes. Section 3.3 presents the domain specific tramway language. Section 3.4 presents a control system for tramway networks and a scenario illustrating its functionality, and finally section 3.5 presents certain simplifications made in the model and implementation.

3.1 Overview

In this section a brief overview of the process of constructing a generic control system is presented.

The informal description presented in chapter 2 covers a wide range of possible network constructions within a certain domain. To capture information about the structure of specific tramways, a domain specific language has been constructed.

A description in the domain specific language consists of two parts: a network description and some interlocking tables. The network description describes the static, physical aspects of the network and the location of hardware devices such as sensors, signals and points. The interlocking tables contains information about possible traversals of the network. A description in the DST-L includes all information about a network which is necessary to construct a control system for that specific network.

The control system is divided into two parts: a generic part and a part containing information about a specific network. The generic part contains generic control components and a control algorithm. The second part contains configuration data for a specific network. The generic part uses the configuration data to control the specific network.

The configuration data for a specific network is generated based on the description of the network and the interlocking tables in the DST-L. To perform the generation a generator is constructed. Given a description it generates the corresponding configuration data.

The process of developing a control system for a specific tramway network is illustrated in figure 3.1.

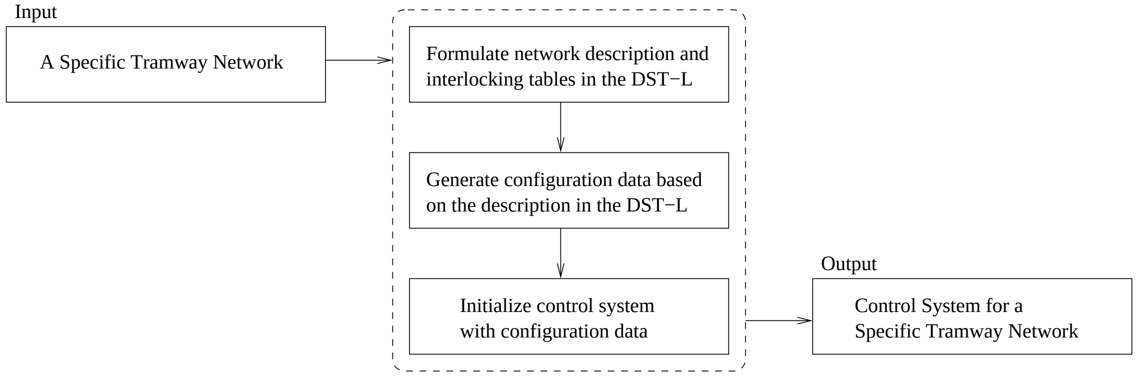

Figure 3.1: Construction of control system.

3.2 Routes

To avoid safety critical situations, it is necessary to be able to control the movements of trams in the network. In this thesis a route based approach to control is used, as described in [HP03b].

A *route* is a continuous path through a network, given by a sequence of sensors which must be passed in order. The *start sensor* of a route is the first sensor in the sensor sequence and must be a sensor next to a border sensor. The *end sensor* of a route is the last sensor of the sensor sequence and must be a border sensor. A signal is placed at the beginning of a route to control traffic entering the route.

A route may pass zero one or more controllable points (cp). For each of these a specific position is required to travel the route.

Conflicting routes Routes may be conflicting. A *conflict* between routes can be either physical or of safety reasons.

If two routes pass the same controllable point, but require two different positions, the routes are conflicting for *physical* reasons.

If two routes include a common sensor, or if two routes pass through the same crossing, the routes are conflicting for *safety* reasons.

Travelling two conflicting routes at the same time could cause a collision. Hence, conflicting routes must not be in use at the same time.

The definition of the two types of conflicts makes it possible for two routes to be conflicting of both safety reasons and of physical reasons.

When reserving a route the control system checks that the *entry signal* of all routes which are safety conflicting with the considered route show *stop*. Therefore, two routes sharing a common entry signal must not be safety conflicting even though they obviously pass a common sensor. Since they certainly are conflicting it is necessary to guarantee that two routes sharing a common entry signal always are conflicting of physical reasons. They are for the following reasons.

Two routes sharing a common entry signal, also share a common start sensor. For the two routes not to be identical, the sensor sequence defining the routes must be distinct. The only possible way to split the routes into different directions is via a controllable point for which they request distinct point settings. Therefore, two routes sharing a common start sensor will always be conflicting of physical reasons.

When allocating a route it is only the start signal of safety conflicting routes for which the entry signal must show stop. This is because any physical conflicting route not sharing a common entry signal with the considered route will always be safety conflicting with the considered route. They are for the following resons:

Two physical conflicting routes not sharing a common entry signal must pass through a common non controllable point from each their branch. This means that the two routes pass a common sensor (the stem sensor of the non controllable point) and the two routes are therefore safety conflicting.

Routes could also be flank protected, broadening the concept of conflicting routes. *Flank protection* is the concept of preventing trams from entering areas in danger of collisions. This would mean that a route had a setting for all points and signals in the network, effectively preventing other trams from passing sensors of the route. This extra safety precaution is standard for railway systems, but not for tramways, and is therefore not considered in this thesis. Nevertheless, the different parts could be modified to support flank protection.

3.3 Domain Specific Tramway Language, DST-L

This section contains an informal description of a domain specific language for tramways called the "Domain Specific Tramway Language" (DST-L). The language consists of two parts, namely a graphical network description and interlocking tables and is originally presented in [HP02].

3.3.1 Network Description

The first part of the DST-L is a graphical *network description* representing the topology of the network elements, their positions and interconnections. The topology agrees with the informal network description from section 2.1.1.

In figure 3.2 an example of a graphical network description is presented. The network is an exit to a maintenance site. The network includes devices such as points, sensors and signals. A symbol explanation is given in figure 3.3. Signals are only visible from one direction. For example signal *S20* is visible from sensor *G20.0* and not from sensor *G20.2*.

The network in figure 3.2 will be used throughout this thesis as a running example, to exemplify properties of the domain.

3.3.2 Interlocking Tables

The second part of the DST-L is *interlocking tables*, with the purpose of fixing the possible routes through the network and the safety conditions that are sufficient to allow driving these routes.

In a description there are four kinds of interlocking tables: a route definition table, a route conflict table, a point position table, and a signal setting table. These tables are presented in the following.

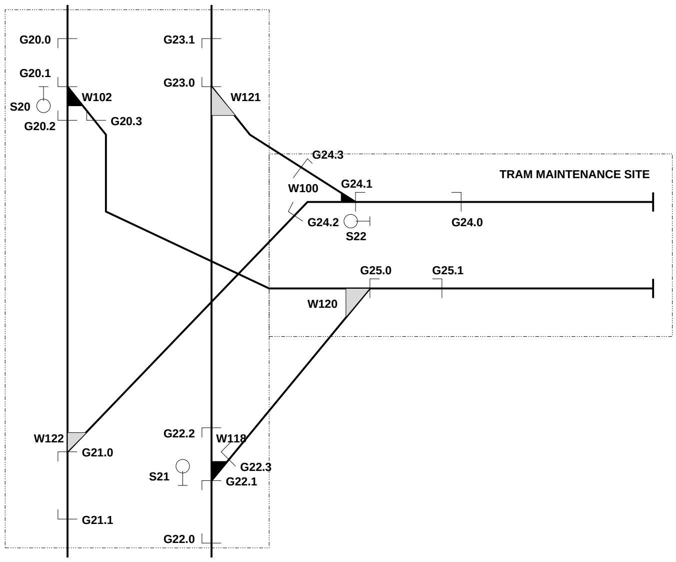

Figure 3.2: Network description in DST-L.

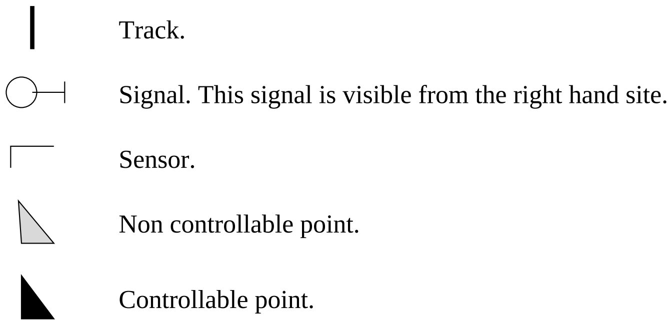

Figure 3.3: Explanation of the symbols used in figure 3.2.

Route Definition Table

The *route definition table* defines the routes in a tramway network. For each route the table contains the route identification and a sensor sequence describing the route. In figure 3.4 an example of a route definition table for the network in figure 3.2 is given.

Route definition table	
Route	**Route Sensor Sequence**
1	<G20.1, G20.2, G21.0, G21.1>
2	<G20.1, G20.3, G25.0, G25.1>
3	<G22.1, G22.2, G23.0, G23.1>
4	<G22.1, G22.3, G25.0, G25.1>
5	<G24.1, G24.3, G23.0, G23.1>
6	<G24.1, G24.2, G21.0, G21.1>

Figure 3.4: Route definition table.

The order in which the routes are presented in the table must match the importance of the routes, as the control system looks at the routes one at a time starting with the one in the first line of the table. This way the routes are given an priority order.

Route Conflict Table

The *route conflict table* contains information about conflicting routes. Physical conflicts are marked with a bullet in the table and conflicts for safety reasons are marked with a circle. In figure 3.5 an example of a route conflict table for the routes in figure 3.4 is given. Due to the conflict property being symmetric, the table is symmetric over the diagonal from the upper left corner to the lower right corner. Please note that the route conflict table contains redundant information as it can be generated from the route definition table and the network description, but nevertheless it is included to make the language more descriptive.

Route conflict table						
Route	**Conflicts with**					
	1	2	3	4	5	6
1		●				○
2	●		○	○		○
3		○		●	○	○
4		○	●			
5			○			●
6	○	○	○		●	

Figure 3.5: Route conflict table.

Point Position Table

For each route the *point position table* contains information of how controllable points along the route should be set in order for the route to be traversable. In figure 3.6 an example of a point position table for the routes in figure 3.4 is given. Like the route conflict table this table is redundant. Please note the obvious lack of flank protection as described in section 3.2.

Point position table			
Route	**W100**	**W102**	**W118**
1	-	STRAIGHT	-
2	-	LEFT	-
3	-	-	STRAIGHT
4	-	-	RIGHT
5	RIGHT	-	-
6	STRAIGHT	-	-

Figure 3.6: Point position table.

Signal Setting Table

For each route the *signal setting table* contains the name of the entry signal and the signal aspect, which the signal must show when the route is open for entering traffic. Like the route conflict and the point position table this table is redundant. In figure 3.7 an example of a signal setting table for the routes in figure 3.4 is given.

Signal setting table	
Route	**Signal Setting**
1	S20:GO-STRAIGHT
2	S20:GO-LEFT
3	S21:GO-STRAIGHT
4	S21:GO-RIGHT
5	S22:GO-RIGHT
6	S22:GO-STRAIGHT

Figure 3.7: Signal setting table.

3.4 Tramway Control System

In this section a *control system* capable of controlling traffic in a tramway network is presented. Furthermore, a scenario illustrating the functionality of the control system is presented. The control system is based on the articles [HP02] and [HP03b].

3.4.1 Description

The control system maintains control of tramways, receiving information from sensors when trams pass by, and requesting specific signal settings and point positions according to certain rules. To keep track of the trams in the network, the control system has a *control counter* for each sensor in the network. When a sensor becomes *high* the corresponding control counter is increased by one. Furthermore the control system has a *route request* and a *route reservation* for each route in the network. The request denotes whether the route is requested by a tram, and the reservation denotes whether the route is reserved.

The main purpose of the control system is the allocation, the allocation closing and the deallocation of routes.

Allocation of a route involves three activities. Before allocating a route it is ensured that no conflicting routes are already reserved. The three activities is as follows:

1. The route is reserved.
2. Points along the route are switched into the proper positions.
3. The entry signal of the route is set to the signal aspect corresponding to the signal setting table.

After the allocation of a route a tram enters. When this happens the start signal of the route is set to *stop* to prevent other trams from entering the route. This is called the *allocation closing*.

Deallocation of a route is performed when a tram has passed the last sensor of a route in use. Deallocation involves two activities:

1. The control counters corresponding to the sensors in the sensor sequence defining the route is set to zero.
2. Removing the reservation of the route.

The control system consists of the following components:

- **Route Dispatcher** The route dispatcher starts allocation of routes. This is based on requests for routes by trams and information from the interlocking tables about which routes are conflicting. This corresponds to the first activity of the route allocation. No route is reserved if a conflicting route is already reserved.

- **Route Controllers** Each route presented in the route definition table has a route controller. The route controller finishes the allocation of the route by setting points along the route, as described by the point position table, and admitting access to the route by setting the entry signal, as described by the signal setting table. The route controller also performs the allocation closing by setting the aspect of the entry signal to *stop*. Furthermore, the route controller deallocates the route when the tram has passed the end sensor.
- **Safety Monitor** The safety monitor constantly checks that certain safety conditions derived from the interlocking tables are met. If not, all signals are set to *stop* to prevent additional problems in the network. Such safety conditions may be violated if tram drivers ignore stop signals or if hardware should fail.
- **Counter Controllers** The counter controllers maintain information of where trams are located on routes. Therefore, the counter controllers have *control counters* counting the number of trams passing a sensor.

3.4.2 A Scenario

To illustrate the operation of a tramway control system a scenario is presented. The scenario is based on the running example presented in section 3.3. In figure 3.8 the network description including routes is presented.

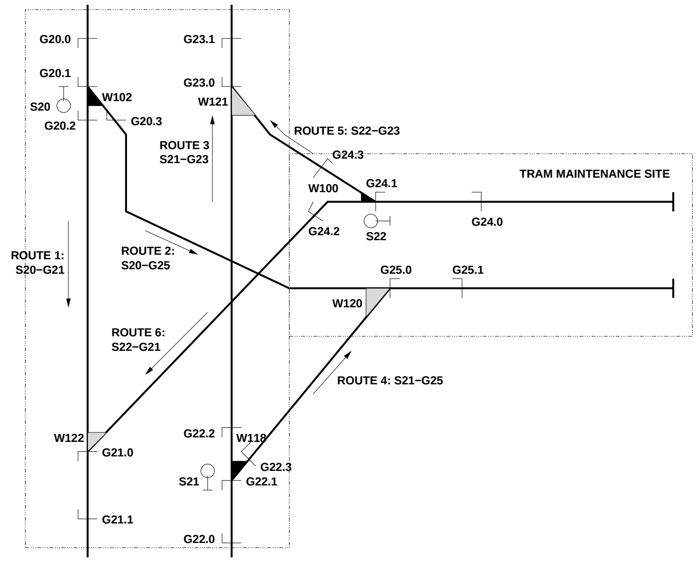

Figure 3.8: The network of the scenario.

Assume that the network is in the initial state, meaning all points are in *straight* position, all signal aspects are *stop* and no routes are requested or reserved.

A tram T1 approaches the network. It passes the sensor `G22.0`. Tram T1 requests route 4 in the network. The route dispatcher receives the request, and checks whether any routes conflicting with route 4 are already reserved. Since this is not the case, the route dispatcher reserves route 4 to tram T1, thereby activating the route controller for route 4. Thereafter, the route controller sets the controllable point `W118` in its turn position and the signal `S21` to *go right*. The driver of tram T1 notices the *go right* signal and makes the tram start moving forward, thereby passing sensor `G22.1`. This informs the route controller that tram T1 has entered route 4. As soon as the tram has entered the route the entry signal is set to *stop*.

Another tram T2 approaches the same network and passes sensor `G20.0`. Tram T2 requests route 2. The route dispatcher receives the request and checks for already reserved conflicting routes. It notices that route 4 is reserved and conflicting. Route 2 remains in requested state and tram T2 remains outside of route 2, at least until route 4 is empty.

A third tram T3 has been at the maintenance site and now wants to leave. It passes sensor `G24.0`, and the control system registers its position. Tram T3 requests route 5. The route dispatcher checks for reserved conflicting routes and finds none. Now route 5 is allocated by reserving the route and activating the corresponding route controller, which sets the controllable point `W100` in its right position, and signal `S22` to *go right*. The driver of tram T3 notices the *go right* signal, and makes the tram start moving forward hereby passing sensor `G24.1`. Now the route controller knows that the tram has entered route 5. The route controller sets the entry signal to *stop*.

If tram T1 had finished route 4 by passing sensor `G25.1`, thereby leaving the network, the route controller would have deallocated the route by removing its reservation and resetting all control counters along the route. Furthermore, when route 4 is finished route 2 can be allocated to tram T2 by activating the corresponding route controller.

3.5 Simplifications

In the rest of this thesis some simplifications have been made in model and implementation, compared to the informal description of the domain given in chapter 2 and the engineering concepts presented in this chapter. In chapter 19 suggestions on how to add the simplifications to the models are presented. The simplifications are as follows:

Signal aspects: The possible signal aspects have been restricted to *go* and *stop*. The aspects also giving the turning direction of the tram are of little use since the tram driver has no influence on the turning direction but only on the speed of the tram. The purpose of the other signal aspects is to inform the driver whether he is turning in a point or continuing straight. This would help the driver estimate a prudent speed through the point, but has been left out to simplify the model.

Onboard computer: The thesis only considers trams with onboard computers for communication with the control system. Alternatively, computers should be placed at the route entry signal. These computers could then be used to issue requests by the tram driver. The control system cannot discern between this way of requesting and an actual onboard computer and therefore they are considered identical in this thesis.
If no computer was present the requests of routes would have to be based on manual interaction with devices by the tram driver. This would most likely include manual setting of points meaning that points would have shared control between the control system and manual interaction. This is not the intention of the system and therefore some kind of computer must be present for issuing route requests.

Only one tram on a route: To simplify the traffic in the network the control system only allows one tram on a route at a time. Normally, this is not the case, but it has been chosen to simplify the model.

Extended driver protocol: Drivers are expected to adhere to a protocol saying that route requests can only be issued while being directly in front of the entry sensor of the requested route. The reason for this protocol is that the control system does not necessarily know in which order trams are placed on a segment. The protocol prevents two kinds of undesirable situations:

- When two or more trams are located in front of an entry signal, the tram furthest away from the signal may actually get its route reservation first. This would result in the signal showing *go*, and the first tram would probably drive on past the signal. This is not a safety issue as the tram would then drive the route requested by the other tram and no conflicting routes would be allocated. But, if the two tram drivers requested distinct routes, it would possibly result in the tram driving another route than it requested or was going to request, which is obviously not desirable.
- A tram requesting a route while positioned another place in the network might accidently thereby let another tram pass a signal.

Time: The aspect of time has not been considered, as it would complicate both models and implementation. The priority has been to construct a reasonable complete working system, excluding details complicating the system unnecessarily.

Safety monitor: The safety monitor has been left out. The safety monitor can be divided into two parts, one using sensor information to control that the system is in a safe state and one based on time. Since time is not included in the model, the last part is not possible to model. The first part is possible based on the model, but it has been chosen not to include this part of the safety monitor in the model, and instead, because of limited resources, concentrate on other aspects. The safety monitor could be added to the model later on.

Failing Devices: When working with mechanical devices such as controllable points there is a risk of failure. This could be if a rock was placed in a point such that it could not shift properly. In the model failing devices is not considered.

3.6 Chapter Summary

The concept of a domain specific tramway language (DST-L) has been introduced, and the design of one such language has been presented. Furthermore, a route based control system, and a scenario illustrating the functionality of the control system has been presented. Finally a number of simplifications have been listed.

Chapter 4

Method Overview

This chapter gives an overview of the method used in this thesis, and of the RSL models.

In the first section of this chapter the development phases of the project are introduced, and in the following sections an overview of each development phase is presented.

The specification of the models is performed in RSL [Gro92].

In the figures in this chapter squares represent schemes. Solid arrows illustrates parameterization of schemes, the source scheme is a parameter of the target scheme. The source of a dashed arrow is a global object and the target is a scheme using the object. Finally dotted boxes with rounded corners illustrates models consisting of one or more schemes. The extension of schemes are illustrated by placing schemes within other schemes. Thus, a scheme that extends another scheme will have this scheme within the box.

4.1 Development Phases

The development in this thesis can be divided into the following phases:

1. **DST-L Modelling** (chapter 5)
   - First an abstract applicative model of the DST-L is constructed.
   - The abstract model is refined to a concrete applicative model.
   - The model of the interlocking tables is further refined.
2. **STS-L Modelling** (chapter 7)
   - The abstract syntax of the STS-L is modelled.
   - The static semantics of the STS-L is modelled.
   - The operational semantics of the STS-L is modelled.
3. **Generator Modelling** (chapter 8)
   The generic control components is given a state transition system semantics.
4. **Safety Requirements Specification** (chapter 9)
   The safety requirements for the tramway system are determined and formalised.
5. **Verification** (chapter 10)
   It is determined how to verify that the safety requirements are fulfilled and an informal argumentation is constructed.
6. **Development of Imperative Models** (chapter 11)
   Imperative models suitable for implementation are constructed for the control system and the generator.

7. **Specification of the Control Algorithm** (chapter 12)
   A control algorithm is specified. The purpose of the control algorithm is to control the overall interpretation of transition rules of the STSs in the control system.
8. **Implementation of the DST-L** (chapter 13)
   The concrete applicative model of the DST-L is implemented in XML.
9. **Implementation of the Control System** (chapter 14)
   The imperative model of the generic part of the control system is implemented as a C program. The program is used for building a simulator of the control system.
10. **Implementation of the Generator** (chapter 15)
    The imperative model of the generator is implemented as an XSL transformation, transforming a description in the DST-L to configuration data.
11. **Implementation of the Simulator** (chapter 17)
    To demonstrate the functionality of the control system, a simulator is implemented. This includes representing the physical environment by STSs.
12. **Testing** (chapter 16)
    The DST-L, the generator, the control system and the simulator are tested.

4.2 Modelling the DST-L

The DST-L facilitates description of tramway networks and interlocking tables within a specific domain. The modelling of the DST-L is performed in two steps described in the following sections.

The model of the DST-L is generic. The model is instantiated based on parameters from a specific network within the domain of the language.

4.2.1 Abstract Applicative Model

The basic *objects* which the DST-L uses to describe tramway systems are the devices in tramway networks, namely sensors, signals and points and the routes defined in a network. The identifiers of these objects constitutes the name space of the DST-L. The identifiers are modelled as sorts in the abstract model.

type ObjectId

Properties of objects are available via observers. Observers are modelled by functions taking one or more identifiers as argument and returning the property of the corresponding object or objects. The observers have function signatures of the form:

value obs: ObjectId $\times$ ... $\rightarrow$ ObjectProp

A derived observer is an observer which is expressed in terms of other observers. A basic observer is an observer with a function signature, but no function definition.

A number of axioms express properties of the observers.

RSL Model

The abstract applicative model is divided into two schemes. One containing the network description and one containing the interlocking tables. The relation between the schemes is illustrated in figure 4.1.

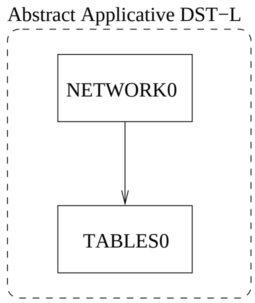

Figure 4.1: Schemes and relations in the abstract applicative model of the DST-L.

4.2.2 Concrete Applicative Model

The abstract model is refined to a concrete applicative model. A super type for identifiers DSTLId is introduced, and object identifiers are concretized to be a subset of this super type.

```
type
  DSTLId,
  ObjectId = {| id:DSTLId • id ∈ elems objectids |}
value
  objectids : DSTLId*
```

The identifiers must be unique expressed by axioms of the form:

```
axiom
  elems objectids_1 ∩ elems objectsids_2 = {}
```

Constants representing the datastructures which are to be implemented are introduced. These contains data about objects.

```
value
  objects: ObjectId → (... × ... × ...) ,
```

The constants makes it possible to concretize the basic observers, since all the necessary data is now available.

```
value
  obs: ObjectId × ... → ObjectProp
  obs(oid, ...) ≡ ...
```

The axioms expresses properties of the observers. Since the signatures of the observers have not changed, the axioms are not changed either.

RSL Model

The relation between the schemes of the concrete applicative model is illustrated in figure 4.2.

4.2.3 Refinement of Concrete Applicative Model

An imperative model of the DST-L is not constructed, but certain imperative models use descriptions in the DST-L and therefore the part of the DST-L model containing the interlocking tables is slightly refined. The resulting model is in scheme TABLES2.

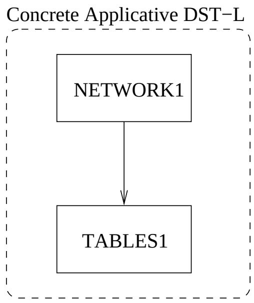

Figure 4.2: Schemes and relations in the concrete applicative model of the DST-L.

4.3 Modelling the STS-L

It is the intention to represent the control components of the control system as state transition systems (STS). Therefore, it is necessary to develop a formal model of an STS-language (STS-L).

An abstract model of the STS-L has not been deemed necessary since the syntax and the semantics of the language are well defined.

The resulting model of the STS-L is generic. It is possible to describe any STS representing control componenets of the control system, and each kind of of device in the physical environment as described in section 4.12.1.

4.3.1 Abstract Syntax

The basic *objects* which the STS-L uses are as for the DST-L the devices in tramway networks, namely, sensors, signals and points. Furthermore a number of control variables is part of the basic objects in the STS-L. The identifiers of these objects constitutes the name space of the STS-L. The identifiers are modelled as the identifiers in the concrete applicative model of the DST-L

type
  STSLId,
  STSLObjectId = {| id : STSLId • id ∈ **elems** STSLobjectids |}
value
  STSLobjectids : STSLId*

The identifiers must be unique expressed by axioms of the form:

axiom
  elems STSLobjectids_1 ∩ **elems** STSLobjectsids_2 = {}

The basic component in the STS-L is the STS. An STS is presented by a number of rules

type
  Sts = Rule*

A rule consists of a number of conditions expressing the necessary state of the system for the rule to fire. Additionally, a rule consists of a number of actions expressing what happens when the rule is fired.

type
  Rule = Condition* × Action*

A condition contains references to objects and/or constants and a relation. Actions contains references to objects and/or constants and an operation.

These are the basic components of the abstract syntax for the STS-L.

4.3.2 Static Sematics

Expressions formulated in the STS-L must meet certain well-formed conditions. These are formulated in the static semantics of the STS-L.

An STS must be well-formed meaning that the rules being part of the STS are well-formed. Rules are well-formed if the conditions and actions of the rule are well-formed. A condition is well-formed if it is possible to use the relation of the condition on the objects of the condition. An action is well-formed if it is possible to use the operation of the condition on the objects of the condition.

To decide which relations and operations can be used on which objects, it is necessary to declare a number of functions deciding the type of objects and different expressions.

```
value
  type_of: ObjectId → Tp
```

4.3.3 Operational Semantics

An STS is interpreted by testing whether one or more of its rules can be fired. A rule is fired if the conditions of the rule evaluates to true. A rule is fired by processing the actions of the rule.

To evaluate conditions it is necessary to access a state space describing the current state of the system. A number of functions is declared looking up data in the state space about specific objects.

```
value
  get_object_data: ObjectId → StateSpace → data
```

It is also necessary to access the state space describing the current state of the system when processing actions. A number of functions is declared modifying the data about a specific object in the state space.

```
value
  set_object_data: ObjectId → StateSpace → StateSpace
```

4.3.4 RSL model

The concrete applicative model consists of the abstract syntax, the static semantics and the operational semantics. The structure of the model is presented in figure 4.3.

4.4 Modelling the Generator

A model of the DST-L and of the STS-L has been constructed. The control components of the control system is represented by STSs. Therefore a model of the concrete STSs representing the control components is constructed.

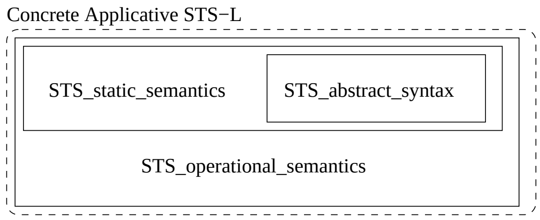

Figure 4.3: Schemes and relations of the concrete applicative model of the STS-L.

The STSs are constructed based on a description in the DST-L. Therefore it is necessary to translate the identifiers from the DST-L to the identifiers of the STS-L. A number of functions facilitate these translations:

value
DSTLId_to_STSLId : DSTLId → STSLId

The model contains a constant representing each concrete STS including all rules, conditions and actions. Since the model is parameterized by the model of the DST-L, it automatically is a generator of the STSs for the control system.

4.4.1 RSL model

The model of the generator is concrete applicative. The structure of the model and its relationship with other models is presented in figure 4.4. The generator is parameterized with the model of the DST-L, the relation to the STS-L is to give the generator access to the syntax of the STS-L. The STS-L is represented by S which is an instance of the applicative model of the STS-L.

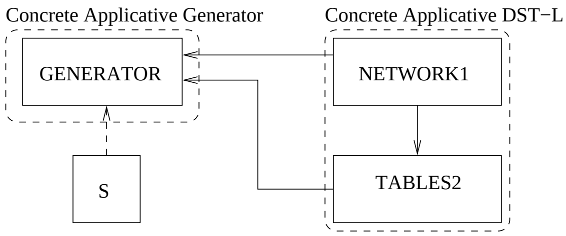

Figure 4.4: Relation between generator model, DST-L model, and STS-L model.

4.5 Safety Requirements Specification

A number of safety predicates is formulated. For a network to be in a safe state these predicates must be fulfilled.

4.6 Verification

To ensure the safety of the network controlled by the control system, it must be verified that actions of the control system cannot make the system unsafe.

This is a matter of proving that all reachable states from the initial state are safe. This can be done in more than one way.

Model Checking All the reachable states are constructed and checked against the safe predicate. This is usually done by means of a tool, but nevertheless the sheer magnitude of constructing all reachable states may cause problems with this method. Furthermore, this method can only be used on a specific instance, meaning that it can only be verified for one instantiation of the global state[1]. A method for proving generic theories is preferred.

Inductive Proofs The method preferred in this project is an inductive proof technique. How to conduct such inductive proofs in RSL is introduced in [HP00].

Basically, the task is to prove that the initial state is safe, and that applying a function modifying the statespace to a safe state yields a safe state. The predicates stating the desired properties of the state space are formulated and the method used for verification is commented on in chapter 10.

4.7 Development of Imperative Models

A development step is performed on the models of the STS-L and the generator. The development transforms the concrete applicative models to concrete imperative models. The structures of the imperative models are close to the actual implementation.

The model of the DST-L is to be implemented in a language which does not benefit from an imperative model and the model is therefore only slightly refined because parts of it is used by an imperative model.

A refinement of the imperative STS-L transforms recursive definitions of functions to iterative definitions.

4.8 Specification of Control Algorithm

An algorithm which maintains the overall control of a tramway network is specified. This algorithm is called the interpreter. It decides the sequence in which the STS control components are interpreted.

4.9 Implementation

The concrete applicative model of the DST-L is implemented using the markup language XML. Furthermore, based on the refinements of the applicative models the generator is implemented using XSLT and the STS-L is implemented using the imperative programming language C.

4.10 Testing

Two types of tests are conducted. These are structural tests, used to test the functionality of the STS-L and the DST-L, and functional tests used to test the DST-L, the generator, and the STS-L. The functional test of the STS-L and the generator is performed by constructing scenarios and checking that the result is as expected.

[1] Symbolic model checking circumvents these problems.

4.11 Instantiation

The running example is described using the DST-L and the generation of configuration code from that description. Thus, the control system can be initialized to control the network represented in the running example. Additionally, examples are initialized utilising network structures not present in the running example.

4.12 Simulation

The running control system is simulated in C using parallel execution of STS control components and the STS simulating the physical environment.

4.12.1 Simulation of the Physical Environment

To simulate the functionality of the physical environment in the tramway network an STS is constructed for each of the physical devices. The transition rules of these STS expresses when a given transition of the state of the device may happen and which state change these events causes. These STSs are generated by the same generator as the STSs representing the control components

In the real world, when a request is given by the control system, the device which the request concerns constantly checks the state space and therefore registers the request and almost instantly changes the state of the device to meet the request. Since the physical devices are represented by STSs, the STSs receives the request and change the state.

4.13 Chapter Summary

In this chapter an overview of the method used in the process of developing the DST-L, the generator and the control system. This includes the approach taken when specifying the formal model, model overview, the verification, the refinement steps, the implementation, the testing, and simulation.

Chapter 5

Model of the Domain Specific Tramway Language, DST-L

In this chapter a model of the domain specific language for tramways is presented. A description in the language consists of a network description and four interlocking tables, as described in section 3.3.

In section 5.1 abstract models of the network description and the interlocking tables are given, and in section 5.2 the abstract models are concretized. Furthermore, in section 5.3 a slightly refined version of the concrete model of the interlocking tables is presented. Finally, the model of the DST-L is compared with traditional models of languages in RSL in section 5.4.

5.1 Abstract Applicative Model

In this section the initial abstract models of the network description and the interlocking tables are described. The models are based on the informal description given in chapter 2.

5.1.1 Network Description

Tramway networks may change over time. Tracks may be added, moved or reordered to meet changes in the requirements. In this model such changes are not considered, therefore, a tramway network is considered static.

The complete abstract model of the network description is in the RSL scheme NETWORK0, found in appendix B.1.1.

To describe a network it is divided into segments. A sort SegmentId is declared which is assumed only to contain identifiers for the segments of the considered network.

type SegmentId

To monitor the traffic in a network sensors are placed at vital locations, and therefore a sort SensorId is declared:

type SensorId

A sensor is located at each end of a segment. The sensors of a segment is defined by the static observer sensors_of:

```
value
  sensors_of : SegmentId → {| ses : SensorId-set • card ses = 2 |}
```

Segments must be uniquely defined by the two sensors:

```
axiom
  [distinct_segments]
    ∀ seg1, seg2 : SegmentId • seg1 ≠ seg2 ⇒ sensors_of(seg1) ≠ sensors_of(seg2),
```

In general a sensor defines an end of two distinct segments. Two segments with an end defined by a common sensor are physically connected. An exception is border sensors, defining the end of a segment at the border of a network. Therefore, border sensors can be identified by the fact that they only define the end of one segment:

```
value
  border : SensorId → Bool
  border(s) ≡ card { seg | seg : SegmentId • s ∈ sensors_of(seg) } = 1
```

It is required that a segment is not defined by two border sensors:

```
axiom
  [double_border_segment]
    ∀ seg : SegmentId •
      let (sen1, sen2) : SensorId × SensorId • {sen1, sen2} = sensors_of(seg) in
        ∼ (border(sen1) ∧ border(sen2))
      end
```

Point

A sort PointId is declared:

```
type PointId
```

The flanking sensors of a point are given by three static observers stem, turn, and straight.

```
value
  stem, turn, straight : PointId → SensorId
```

The three sensors must be distinct to form a junction, and the segments defined by the stem and turn sensor and the stem and straight sensor must exist in the network:

```
axiom
  [distinct_point_sensors]
    ∀ p : PointId • card {stem(p), turn(p), straight(p)} = 3,

axiom
  [point_segments]
    ∀ p : PointId •
      (
       ∃ s1, s2 : SegmentId •
         sensors_of(s1) = {stem(p), turn(p)} ∧ sensors_of(s2) = {stem(p), straight(p)}
       )
```

The derived static observer point_at determines whether a sensor is a stem sensor for a point, and if so, returns the identifier for the point.

```
type
  PointOpt == none | mk_PointId(poi : PointId)

value
  point_at : SensorId → PointOpt
  point_at(s) ≡
    if ∃ p : PointId • stem(p) = s then
      let p : PointId • stem(p) = s in mk_PointId(p) end
    else
      none
    end
```

This observer is deterministic due to an axiom, [distinct_stem], stating that at most one point has a given sensor as stem sensor.

A point can be controllable. This refers to the dynamic aspect that the position of a point can be changed, but in this step only the static property that a point can be controllable, is modelled. The property is given by the static basic observer controllable:

```
value
  controllable : PointId → Bool,
```

Crossing

Crossings are modelled by a static observer crossing, determining whether a pair of segments are crossing:

```
value
  crossing : SegmentId × SegmentId → Bool
```

The observer is symmetric:

```
axiom
  [crossing_symmetric]
    ∀ s1, s2 : SegmentId • crossing(s1, s2) ≡ crossing(s2, s1)
```

It is assumed that frontal collisions of two trams driving on the same segment are possible if the segment crosses itself, whether the trams are driving in opposite directions or the same direction. Therefore, a segment must not cross itself:

```
axiom
  [not_self_crossing]
    ∀ s : SegmentId • ∼ crossing(s, s)
```

Furthermore, crossing segments should not include a border sensor, as segments at a border are not always a part of a route. Therefore, it is not controlled that trams do not collide, since control is only possible on routes:

```
axiom
  [no_border_crossing]
    ∀ seg1, seg2 : SegmentId, sen : SensorId •
      sen ∈ sensors_of(seg1) ∧ border(sen) ∧ seg1 ≠ seg2 ⇒ ∼ crossing(seg1, seg2),
```

Signal

To control traffic, signalling devices are placed at certain locations in a network. Thus, a sort SignalId is declared:

```
type SignalId
```

The location of a signal is abstracted by a sensor in the network, given by a basic observer:

```
value
  sensor_of : SignalId → SensorId
```

Hence, a sensor may have an associated signal. This is modelled by a variant type SignalOpt and a derived static observer signal_at returning the signal at a sensor, if any.

```
type
  SignalOpt == none | mk_SignalId(sig : SignalId)

value
  signal_at : SensorId → SignalOpt
  signal_at(s) ≡
    if ∃ sig : SignalId • sensor_of(sig) = s then
      let sig : SignalId • sensor_of(sig) = s in mk_SignalId(sig) end
    else
      none
    end
```

The observer is deterministic since at most one signal is placed at each sensor, stated in the axiom [distinct_signal_sensors].

Well-formed Network

Networks must comply with a number of constraints, to be considered well-formed.

It should be possible to perform an acyclic traversal of a network. Therefore, a network must have at least two border sensors:

```
axiom
  [min_border_sensor]
    card { sen | sen : SensorId • border(sen) } ≥ 2
```

Sensors that are not at a border or at the stem of a point must define an end of exactly two segments:

```
axiom
  [linear_connectivity]
    ∀ sen : SensorId •
      ∼ border(sen) ∧ point_at(sen) = none ⇒
      card { seg | seg : SegmentId • sen ∈ sensors_of(seg) } = 2
```

If a sensor is the stem sensor of a point, it must define an end of exactly three segments:

```
axiom
  [stem_connectivity]
    ∀ sen : SensorId •
      point_at(sen) ≠ none ⇒ card { s | s : SegmentId • sen ∈ sensors_of(s) } = 3
```

Please note that this axiom also implies that a stem sensor cannot be a border sensor. This is intended since it is not possible to control points outside routes.

A number of special rules defining how junctions must be connected is stated in section 2.4.2. These are expressed in the model by the axiom [point_connectivity]:

```
axiom
  [point_connectivity]
   ∀ p1, p2 : PointId •
     p1 ≠ p2 ⇒
     let
       pss = ({stem(p1), straight(p1), turn(p1)} ∩ {stem(p2), straight(p2), turn(p2)})
     in
       card pss ≤ 1 ∨ {straight(p1), turn(p1)} = {straight(p2), turn(p2)}
     end
```

In section 2.4.2 it was also stated that two junctions cannot have the same stem sensor:

```
axiom
  [distinct_stem]
   ∀ p1, p2 : PointId • p1 ≠ p2 ⇒ stem(p1) ≠ stem(p2)
```

The location of signals in a network are restricted to certain positions. The main purpose of signals is to control the traffic entering the network. Thus, they must be placed at a sensor which jointly defines a segment together with a border sensor:

```
axiom
  [signal_placement]
   ∀ sig : SignalId •
     ∼ border(sensor_of(sig)) ∧
     ( ∃ seg : SegmentId, sen : SensorId •
       sensors_of(seg) = {sen, sensor_of(sig)} ∧ border(sen) )
```

The direction of a signal has been decided to be such that the signal is visible when approaching from the border sensor next to it. Thus, a signal placed at a sensor has only one possible direction. Therefore, the direction will not be identifiable in a network consisting of only two segments, since the sensor of the signal would be flanked by two border sensors. To prevent this situation, it is required that two segments both defined by a border sensor cannot be connected.

```
axiom
  [min_network_size]
   ∀ seg1, seg2 : SegmentId, sen1, sen2 : SensorId •
     seg1 ≠ seg2 ⇒
     ( sen1 ∈ sensors_of(seg1) ∧
       sen2 ∈ sensors_of(seg2) ∧ border(sen1) ∧ border(sen2) ⇒
       sensors_of(seg1) ∩ sensors_of(seg2) = {} )
```

If two signals were located at the same sensor, they would have the same direction, due to the definition of the direction of signals just introduced. Hence, in this model signals at the same position with different directions are not considered, even though it might be relevant in some real world situations. Therefore, no two signals must be placed at the same sensor:

```
axiom
  [distinct_signal_sensors]
   ∀ sig1, sig2 : SignalId • sig1 ≠ sig2 ⇒ sensor_of(sig1) ≠ sensor_of(sig2),
```

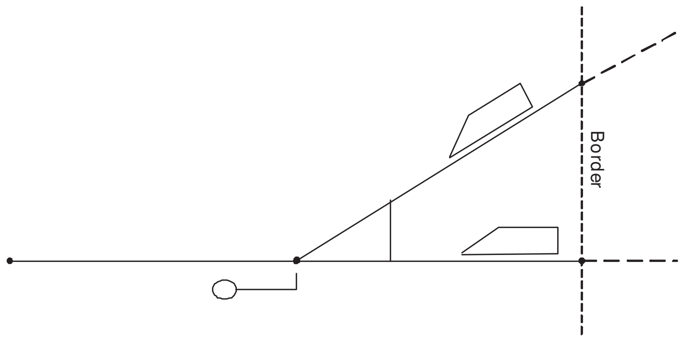

Figure 5.1: Two trams approaching point with signal.

A signal placed at a stem sensor must only be visible from the stem side. If a signal is visible from for instance the turn branch, it is also most likely to be visible from the straight branch. In this case a tram driver might think that the point was set correctly for him to continue, even though this was not the case, resulting in a possible collision. To avoid this, two axioms are introduced stating that neither turn nor straight sensor of a point must be a border sensor (see figure 5.1). Combined with the definition of the visibility of signals this prevents the above mentioned situation:

axiom
  [turn_not_border] $\forall$ pid : PointId $\bullet \sim$ border(turn(pid)),
  [straight_not_border] $\forall$ pid : PointId $\bullet \sim$ border(straight(pid))

5.1.2 Interlocking Tables

The definition of routes and their properties are given by interlocking tables. These tables are described informally in section 3.3.2. Nevertheless, this section describes an abstract model, parameterized with a network description, with static observers defining the properties of each route.

This model has been developed to perform the same development process as for the model of the network description.

Since it is known what information is stored in each table, even though the actual tables are not introduced, this section is divided into paragraphs representing the tables.

The complete abstract model of the routes is given by the RSL scheme TABLES0, found in appendix B.2.1.

Route Definition Table

A sort RouteId is declared to identify the routes:

type RouteId

A route is modelled as a list of sensors which the tram has to pass in sequence to drive the route. This sensor list is given by the static observer route_sensors_of.

value
  route_sensors_of : RouteId $\rightarrow$ N.SensorId*

The sequence of sensors defining a route must be unique, such that no two routes can be defined by the same list of sensors:

```
axiom
  [distinct_routes]
    ∀ rid1, rid2 : RouteId •
      rid1 ≠ rid2 ⇒ route_sensors_of(rid1) ≠ route_sensors_of(rid2),
```

Routes must start at a sensor which is associated to a signal. This is to ensure that access to the route can be controlled. The function route_start returns the start sensor of the route which is the first sensor in the sensor list defining the route:

```
axiom
  [route_start_signal]
    ∀ rid : RouteId • N.signal_at(route_start(rid)) ≠ N.none,
```

A route should denote a traversable sensor sequence in the network. Therefore, two succeeding sensors in the list of a route must constitute a physical path in the network. A physical path between two sensors s1 and s2 exists[1] if they jointly define a segment and if one of the following cases of the observer path is fulfilled:

- Neither of the sensors are the stem sensor of a point.

```
card { p | p : N.PointId • N.stem(p) ∈ {s1, s2} } = 0
```

- s1 is the stem sensor of a non-controllable point and s2 is straight sensor.

```
∃ p : N.PointId • ∼N.controllable(p) ∧ s1 = N.stem(p)  ∧ s2 = N.straight(p)
```

- s1 is the straight sensor or the turn sensor of a non-controllable point, and s2 is the stem sensor.

```
∃ p : N.PointId •
  ∼N.controllable(p) ∧  s2 = N.stem(p) ∧ s1 ∈ {N.turn(p), N.straight(p)}
```

- s1 is the stem sensor of a controllable point and s2 is straight sensor or turn sensor.

```
∃ p : N.PointId •
  N.controllable(p) ∧ s1 = N.stem(p) ∧  s2 ∈ {N.straight(p), N.turn(p)}
```

- s1 is the straight or turn sensor of a controllable point and s2 is the stem sensor.

```
∃ p : N.PointId •
  N.controllable(p) ∧ s2 = N.stem(p) ∧  s1 ∈ {N.turn(p), N.straight(p)}
```

- s1 is the stem sensor of a point, and s2 is not affiliated[2] with any other point. Or, s2 is the stem sensor of a point, and s1 is not affiliated with any point.

```
∃ p : N.PointId •
  (
    N.stem(p) = s1 ∧
    ∼ (∃ p2 : N.PointId • s2 ∈ {N.stem(p2), N.turn(p2), N.straight(p2)})
  ) ∨
  (
    N.stem(p) = s2 ∧
    ∼ (∃ p2 : N.PointId • s1 ∈ {N.stem(p2), N.turn(p2), N.straight(p2)})
  )
```

[1]Note that even though a path exists from s1 to s2 the opposite is not necessarily the case, because of the possible ways to traverse a point.

[2]Not stem, straight, or turn sensor for any point

- The two sensors are point sensors of two distinct points.

```
∃ p1, p2 : N.PointId • p1 ≠ p2 ∧
  {N.stem(p1), N.turn(p1), N.straight(p1)} ∩
  {N.stem(p2), N.turn(p2), N.straight(p2)} = {} ∧
  s1 ∈ {N.stem(p1), N.turn(p1), N.straight(p1)} ∧
  s2 ∈ {N.stem(p2), N.turn(p2), N.straight(p2)}
```

Thus, it is possible to express the requirement that a route must be a path in the network. This is given by the axiom [route_is_path] which recursively calls path by the function path_recursion:

```
axiom
  [route_is_path]
   ∀ rid : RouteId • path_recursion(route_sensors_of(rid))
```

Additionally, routes must end at a border sensor, ensuring that once the tram has left a route it has also left the network. Otherwise, trams can traverse sections of the network without any form of control. The function route_end returns the end sensor of a route which is the last sensor in the sensor list defining the route:

```
axiom
  [route_end_border] ∀ rid : RouteId • N.border(route_end(rid))
```

Routes must, according to section 2.4.1, traverse junctions in specific directions. This corresponds to traversing the point of the junction in a specific direction. The restriction of traversing junctions in this way facilitates the intended purpose of the two types of points related to the junctions. This is given by the axiom [point_direction_of]. This is not to be confused with the path predicate denoting the possible physical path between two sensors.

```
axiom
  [point_direction_of]
    ∀ rid : RouteId •
      let sl = route_sensors_of(rid) in
        ∀ i : Nat • i ∈ inds sl ⇒
          case N.point_at(sl(i)) of
            N.mk_PointId(p) →
            ...
```

If a sensor in the list defining a route is a stem sensor of a controllable point, the following sensor in the list must be either the turn or the straight sensor of that point, and the preceding sensor must not be the turn or the straight sensor of that point:

```
            ...
              if N.controllable(p) then
                ((i < len sl) ⇒ sl(i + 1) ∈ {N.turn(p), N.straight(p)}) ∧
                ((i > 1) ⇒ sl(i − 1) ∉ {N.turn(p), N.straight(p)})
              else
                ...
```

If a sensor in the list is a stem sensor of a non-controllable point, the legal traversal of the junction is opposite:

```
                ...
                ((i < len sl) ⇒ sl(i + 1) ∉ {N.turn(p), N.straight(p)}) ∧
                ((i > 1) ⇒ sl(i − 1) ∈ {N.turn(p), N.straight(p)})
```

```
                end,
              N.none → true
            end
        end
```

Since segments have no definition of direction, a network is not directed in any way. The restrictions on the possible ways for routes to traverse points introduce a kind of direction. All segments connected, not necessarily directly, with a junction will only be traversed in one direction, namely, the one corresponding to the allowed direction through the junction. It is therefore possible to construct a directed network by placing controllable and non-controllable points at the right locations, and demanding that the network is covered by routes and that trams only drive on routes.

Since it is only routes which are given a direction, it would be a problem if a junction with a controllable point was located at the border of the network. Then it would not be possible to control the tram direction through the junction. But since neither stem, turn nor straight sensors can be border sensors, according to the axioms [stem_connectivity], [turn_not_border], and [straight_not_border] this is not a possible construction.

A route must be acyclic:

```
axiom
  [acyclic_routes]
   ∀ rid : RouteId •
     len route_sensors_of(rid) = card elems route_sensors_of(rid)
```

Furthermore, it is stated in the axiom [all_borders_entry_exit] that every border sensor is either an entry border sensor or an exit border sensor. Thus, all entrances to the network are route ends or jointly with a route start define a segment.

Route Conflict Table

Whether two routes are conflicting or not is a static property, describing whether the risk of collisions exists if there were trams driving each route simultaneously, or if it is not possible to allocate both routes at the same time (see section 3.3.2). This property is modelled by the static basic observers conflicts_physical_with and conflicts_safety_with connecting each route to the set of conflicting routes:

```
value
  conflicts_physical_with : RouteId → RouteId-set,
  conflicts_safety_with : RouteId → RouteId-set,
```

This definition of the possible conflicts can be further expanded. If two routes are conflicting for safety reasons it can either be because they pass the same sensor, or because they are crossing. Thus, the properties of the two basic observers can be declared in three axioms:

```
axiom
  [passing_same_sensor_conflict]
   ∀ rid1, rid2 : RouteId •
     passing_same_sensor(rid1, rid2) ∧ route_start(rid1) ≠ route_start(rid2) ⇒
     rid1 ∈ conflicts_safety_with(rid2),

  [crossing_routes_conflict]
   ∀ rid1, rid2 : RouteId •
```

```
      routes_crossing(rid1, rid2) ∧ route_start(rid1) ≠ route_start(rid2) ⇒
      rid1 ∈ conflicts_safety_with(rid2),

  [point_position_conflict]
    ∀ rid1, rid2 : RouteId •
      point_conflict(rid1, rid2)≡rid1 ∈ conflicts_physical_with(rid2)
```

Thus, the observers conform to the fact that the route conflict table can be generated from the route definition table and the network description, making sure that the redundant data does not introduce errors. The axioms [passing_same_sensor_conflict] and [routes_crossing_conflict] do not consider routes with common route starts, as the conflict will then be caught by point_conflict (see section 3.2):

```
value
  routes_crossing : RouteId × RouteId → Bool
  routes_crossing(rid1, rid2)≡
    let sl1 = route_sensors_of(rid1), sl2 = route_sensors_of(rid2) in
      ( ∃ s1, s2 : N.SegmentId •
        (∃ i : Int • i > 0 ∧ i < len sl1 ∧ N.sensors_of(s1) = {sl1(i), sl1(i + 1)}) ∧
        (∃ i : Int • i > 0 ∧ i < len sl2 ∧ N.sensors_of(s2) = {sl2(i), sl2(i + 1)}) ∧
         N.crossing(s1, s2) )
    end,

  passing_same_sensor : RouteId × RouteId → Bool
  passing_same_sensor(rid1, rid2)≡
    elems route_sensors_of(rid1) ∩ elems route_sensors_of(rid2) ≠ {}

  point_conflict : RouteId × RouteId → Bool
  point_conflict(rid1, rid2)≡
    ( ∃ pid : N.PointId •
      pid ∈ points_at(rid1) ∩ points_at(rid2) ∧
      pointsetting_of_point_in_route(pid, rid1) ≠
      pointsetting_of_point_in_route(pid, rid2) )
```

Please note that the observers points_at and pointsetting_of_point_in_route determine points in a route, and the pointsetting of those, respectively. These observers are explained in the paragraph below considering the Point Position Table.

A derived observer conflicting determines whether two routes are conflicting.

```
value
  conflicting : RouteId × RouteId → Bool
  conflicting(rid1, rid2)≡
    rid2 ∈ (conflicts_safety_with(rid1) ∪ conflicts_physical_with(rid1))
```

It is enforced that the observer is symmetric:

```
axiom
  [symmetric_conflicts]
    ∀ rid1, rid2 : RouteId • conflicting(rid1, rid2) ⇒ conflicting(rid2, rid1)
```

Reflexivity has not been enforced on conflicting observer, since this property is not necessary. The main purpose for introducing reflexivity would be to prevent two trams from driving the same route simultaneously. But, two trams driving the same segment in the same direction are not considered an unsafe situation, and thus reflexivity is not enforced. Furthermore, the control system as it is designed (see chapter 6), does not allow two trams to be on the same route. This is mainly to simplify the system.

Point Position Table

Along each route zero, one, or more points are located. These are either controllable or non-controllable. The controllable points along a route are given by the static observer points_at:

```
value
  points_at : RouteId → N.PointId-set,
```

For a tram to be able to follow the sensors defining a route, the controllable points along the route must be set in certain positions.

A sort PointSetting is declared to represent the possible settings of a point:

```
type PointSetting
```

The necessary setting of the controllable points along a route is given by the static partial observer pointsetting_of_point_in_route. The observer is partial, since not all points should be given a setting.

```
value
  pointsetting_of_point_in_route : N.PointId × RouteId →̃ PointSetting
```

The enforcement of the setting of points on routes are given by the axiom [ppt_entries]. The observers pointsetting_is_straight and pointsetting_is_turn determines if the setting is straight or turn, respectively.

```
axiom
  [ppt_entries]
   ∀ rid : RouteId •
     let sl = route_sensors_of(rid) in
       ∀ i : Nat • i ∈ inds sl ⇒
         case N.point_at(sl(i)) of
           N.mk_PointId(p) →
             if N.controllable(p) then
               p ∈ points_at(rid) ∧
               if sl(i + 1) = N.straight(p) then
                 pointsetting_is_straight(pointsetting_of_point_in_route(p, rid))
               else
                 pointsetting_is_turn(pointsetting_of_point_in_route(p, rid))
               end
             else
               true
             end,
           N.none → true
         end
     end
```

Signal setting table

To enter a route the entry signal must be set to the right signal aspect. It is therefore necessary to be able to express the settings of a signal. Hence, a sort SignalSetting is declared to represent the possible signal settings:

```
type SignalSetting
```

The signal of a route is given by the static observer signal_of.

```
value
  signal_of : RouteId → N.SignalId
```

This signal must be placed at the entrance to the route (start sensor of the route):

```
axiom
  [sst_entries] ∀ rid : RouteId • route_start(rid) = N.sensor_of(signal_of(rid))
```

A basic observer signalsetting_of_signal_at returns the wanted signalsetting of the entry signal for route allocation.

```
value
  signalsetting_of_signal_at : RouteId → SignalSetting
```

An axiom [signal_setting_correct] is declared to express if the setting of the signal is correct. The basic observer signal_setting_is_go tells whether a signalsetting is go.

```
axiom
  [signal_setting_correct]
    ∀ rid : RouteId • signalsetting_is_go(signalsetting_of_signal_at(rid))
```

5.2 Concrete Applicative Model

In this section the model resulting from concretizing the initial abstract model is presented. The purpose of the refinement has been to make the sorts and basic observers concrete.

In section 5.2.1 the concrete network description model is presented, and in section 5.2.2 the concrete interlocking tables model is presented.

Only the parts that have been changed in the refinement are presented.

5.2.1 Network Description

The concrete model of the network description is in the RSL scheme NETWORK1 which can be found in appendix B.1.2.

The sorts presented in the initial abstract model of the network description (see section 5.1) have been made concrete.

Identifiers of devices and segments have been concretized by introducing a sort DSTLId functioning as a super type of identifiers:

```
type DSTLId
```

The collection of names for each of the devices are captured in DSTLId-lists. E.g. the names of the sensors are given by the list sensorids.

```
value
  sensorids : DSTLId*
```

Hence, the sort SensorId can be concretized as a sub-type of DSTLId, namely, those DSTLId that are present in sensorids.

```
type
  SensorId = {| id : DSTLId • id ∈ elems sensorids |},
```

The sorts PointId, SignalId, and SegmentId are concretized similarly. To ensure that an identifier is not an identifier for two different types of devices or device and segment, the axiom distinct_dstl_ids is introduced.

```
axiom
  [distinct_dstl_ids]
    elems sensorids ∩ elems pointids = {} ∧
    elems sensorids ∩ elems signalids = {} ∧
    elems sensorids ∩ elems segmentids = {} ∧
    elems pointids ∩ elems signalids = {} ∧
    elems pointids ∩ elems segmentids = {} ∧
    elems signalids ∩ elems segmentids = {}
```

Furthermore, it is assured that the lists of identifiers do not contain duplicates :

```
axiom
  [no_duplicates_in_sensorids] card elems sensorids = len sensorids,

  [no_duplicates_in_pointids] card elems pointids = len pointids,

  [no_duplicates_in_signalids] card elems signalids = len signalids,

  [no_duplicates_in_segmentids] card elems segmentids = len segmentids
```

To concretize the basic observers a number of tables have been introduced, containing the network description of the tramway language. Each table is modelled as an RSL function, to ensure that all identifiers have entries. Hence, all devices and segments have entries in the tables, enabling looking up information by name. Since the only relevant information about sensors is the names, it is ensured by the axiom [sensor_total] that all identifiers for sensors are present in the network description.

```
value
  points : PointId → ((SensorId × SensorId × SensorId) × Bool),
  segments : SegmentId → (SensorId × SensorId × SegmentId-set),
  signals : SignalId → SensorId,
  sensors : SensorId-set

axiom
  [sensor_total] sensors = elems sensorids,
```

As a consequence of the declaration of the network description given above, all basic observers can be specified explicitly. For instance the observer sensors_of is defined in terms of segments:

```
value
  sensors_of : SegmentId → {| ses : SensorId-set • card ses = 2 |}
  sensors_of(s) ≡ let (sen1, sen2, segs) = segments(s) in {sen1, sen2} end
```

5.2.2 Interlocking Tables

The concrete model of the interlocking tables is in the RSL scheme TABLES1 which can be found in appendix B.2.2.

The development step is similar to the one presented for the network description in section 5.2.1. The main difference is the knowledge of the end product, being some kind of representation of the interlocking tables described in section 3.3.2.

The sort RouteId is concretized similarly to the device names (see section 5.2.1), as a sub-type of the DSTLId type. Similarly, the fact that a route name is not also the name of some device is in the axiom distinct_dstl_ids2 :

```
type
  RouteId = {| id : DSTLId • id ∈ elems routeids |}

axiom
  [distinct_dstl_ids2]
    elems N.sensorids ∩ elems routeids = {} ∧
    elems N.signalids ∩ elems routeids = {} ∧
    elems N.pointids ∩ elems routeids = {} ∧
    elems N.segmentids ∩ elems routeids = {}
```

Additionaly, it is ensured that the list route identifiers does not contain duplicates:

```
axiom [no_duplicates_in_routeids] card elems routeids = len routeids
```

The sorts PointSetting and SignalSetting have been concretized as variant type definitions, enumerating the values presented in the interlocking tables presented in section 3.3.2. Only the following values have been modelled, according to the simplifications given in section 3.5.

```
type
  PointSetting == TURN | STRAIGHT,
  SignalSetting == GO | STOP
```

To concretize the basic observers a number of tables have been introduced. This method is similar to the one presented for the network description in section 5.2.1.

Thus, the interlocking tables are modelled as functions:

```
value
  rdt : RouteId → N.SensorId*,
  rct : RouteId → RouteId-set × RouteId-set,
  ppt : RouteId → (N.PointId →m PointSetting),
  sst : RouteId → (N.SignalId × SignalSetting)
```

As a consequence of these functions each basic observer can be defined explicitly in terms of the interlocking tables functions. For instance the observer route_sensors_of is defined in terms of the route definition table (rdt):

```
value
  route_sensors_of : RouteId → N.SensorId*
  route_sensors_of(rid) ≡ rdt(rid)
```

Development Relation

```
devt_relation NETWORK0_1 ( NETWORK1 for NETWORK0 ) :
 ⊢ NETWORK1 ≼ NETWORK0

devt_relation TABLES0_1 ( TABLES1 for TABLES0 ) :
 in class object N : NETWORK1 end ⊢
  ( ⊢ TABLES1(N) ≼ TABLES0(N))
```

NETWORK1 and TABLES1 are an implementation of NETWORK0 and TABLES1 respectively as the only difference is the extension of signature definitions, and addition of signatures.

5.3 Refinement of Concrete Applicative Model

The concrete model of the interlocking tables has been refined in the RSL scheme TABLES2 found in appendix B.2.3. This is done to ease the specification and the implementation of the generator.

In the first concrete model the interlocking tables are modelled using sets. The generator iterates on the different tables, and thus will encounter problems as the iteration of sets is generally not deterministic.

Thus, a refined version of the route conflict table (rct) and the point position table (ppt) using lists have been introduced:

```
value
  rct_refined : RouteId → RouteId* × RouteId*,
  ppt_refined : RouteId → (N.PointId × PointSetting)*
```

It is ensured that the tables do not contain duplicates:

```
axiom
  [no_duplicates_rct_refined]
    ∀ rid : RouteId •
      let (l1, l2) = rct_refined(rid) in
        card elems l1 = len l1 ∧ card elems l2 = len l2
      end,

  [no_duplicates_ppt_refined]
    ∀ rid : RouteId •
      let pointlist = points_in_ppt_ref(rid) in
        len pointlist = card elems pointlist
      end
```

Please note that the derived observer points_in_ppt_refined returns the list of points for a route.

The tables introduced in section 5.2 can be derived from the refined tables:

```
value
  rct : RouteId → RouteId-set × RouteId-set
  rct(rid)≡let (pcl, scl) = rct_refined(rid) in (elems pcl, elems scl) end,

  ppt : RouteId → (N.PointId →m PointSetting)
  ppt(rid)≡
    [ pid ↦ ps | pid : I1.PointId, ps : PointSetting • (pid, ps) ∈ elems ppt_refined(rid)]
```

Development Relation

```
devt_relation TABLES1_2 ( TABLES2 for TABLES1 ) :
 in class object N : NETWORK1 end ⊢
  ( ⊢ TABLES2(N) ≼ TABLES1(N) )
```

This development is an implementation, as the only difference is the extension of signature definition, and addition of signatures.

5.4 Comparison With Traditional Language Development

The method used to develop the model of the DST-L differs from the standard approach. The abstract syntax is initially formulated in an abstract applicative style, and the static semantics are formulated axiomatically, through abstract basic observers. The operational semantics is not present in the DST-L and therefore, not formulated.

In the model a network is represented by the NETWORK scheme. The identifiers in the scheme are all related to the specific network represented by the scheme. This means that in an instantiation of the model all objects must be part of the network.

Modelling the language in the traditional RSL style, would result in a Network type. Introducing a Network type it is necessary to reevaluate the concept of identifiers. With a Network type the model only makes sense if the identifiers are global meaning that the set of identifiers not only contains identifiers for a specific network but for several networks.

Introducing a Network type would also lead to an extra parameter for each observer. This parameter would identify a specific network. Because of the global identifiers each observer would be partial since a pre-condition is necessary to ensure that the object corresponding to the argument identifier actually is part of the argument network of the observer. The same considereations would occur with a Tables type.

Partial functions are not desirable when arguing about the safety properties of the model and therefore this untraditional method has been chosen.

5.5 Chapter Summary

In this chapter the RSL model of the DST-L has been presented, including the refinement from abstract to concrete model, and the development has been compared with traditional language development in RSL.

Chapter 6

Control Components and Physical Environment

In this chapter the components of the control system and the physical environment of the network are given an STS semantics and a control algorithm is presented.

First, a notion of a global state for the system is presented in section 6.1. Thereafter, the different STSs are presented, informally and more formally by state transition diagrams in sections 6.2 - 6.4, and finally a control algorithm, the interpreter, is presented in section 6.4.4.

The STS control components presented in this chapter are based on [Hax03].

For a short introduction to STS syntax please refer to appendix A.

6.1 Global State Space

The global state space contains the current state of the control components and the physical environment, which are the values of both control and physical variables. The control components are the route dispatcher, the route controllers, and the counter controllers. The physical environment consists of the different devices.

- *Control Variables*[1].
  - For each sensor a related control counter exists. The control counter is used by the control system to decide where in the network trams are located. See section 6.3.1.
  - Requested setting of devices (only signals and points). Since changing the state of a device can be a mechanical or electrical process, the control system cannot directly set it. Therefore, the requested state of devices in the state space represents requests issued by the control system.
  - For each route two *route control variables (rcv)* exist. One represents whether the route is requested and the other whether the route is reserved
- Physical Variables[2].
  - The actual (physical) setting of devices (signals, points, sensors).

[1] Variables in the control system
[2] Variables in the devices

  - The values of sensor counters, used to simulate sensors. See section 6.4.
  - The destination of the sensors, being the sensor the last tram that passed the sensor is driving towards. See section 6.4.
- Control Mode Variables. The control mode variables are present in both the control components and the physical environment. Each represent the current state of an STS.

The sensor counter and the destination sensor are auxiliary variables. Sensor counters are used to express the safety predicates, but is also used in the simulation. Destination sensors are only used to express the safety predicates.

6.2 STSs Representing the Global System

Both the physical environment (NTS), and the control components (CTS) are given an STS semantics, as suggested in [HP03a] and [HP03b].

This leads to the STS given by parallel combination between the physical environment and the control components, representing the entire system:

$$(NTS \parallel CTS)$$

This STS acts on the global state space introduced. The STS control components read physical variables, and write control variables. The physical environment reads control variables and writes physical variables.

To give a graphical presentation of the STS, the notation of state transition diagrams is used. Using this notation the conditions and actions involving the control mode variables are given by the graphical representation of states.

Likewise, in the descriptions the conditions and actions concerning the control mode variables are not included, but it is intuitively understood that for the transition rule from a state 1 to a state 2 to fire, the control mode variable must have the value of state 1, and the resulting value of the control mode variable is the value of state 2.

The STSs are not necessarily deterministic and therefore the transition rules are given priority deciding the order of processing. In the state transition diagram this order is given by numbers on the transitions, with 1 denoting the highest priority.

The table in figure 6.1 explains the notation of variables used in the state transition diagrams.

6.3 STS Control Components

The control components of the control system is a parallel combination of STSs modelling the behavior of each of the components:

$$CTS = (RD \parallel RC_1 \parallel ... \parallel RC_r \parallel CC_1 \parallel ... \parallel CC_c)$$

where r is the number of routes and c is the number of counters (sensors). RD represents the STS modelling the route dispatcher, RC_i $(0 < i < r)$ the route controller for route i, CC_j $(0 < j < c)$ the counter controller for counter j, as presented in the description of the control system (see section 3.4 for descriptions).

variable	*description*
req(R)	value indicating whether route R is requested
res(R)	value indicating whether route R is reserved
a(S)	the actual (physical) value of signal S
a(W)	the actual (physical) value of point W
r(S)	the requested (control variable) value of signal S
r(W)	the requested (control variable) value of point W
v(G)	value (physical) of sensor G
c(G)	value of counter of sensor G
cc(G)	value of control counter of sensor G
d(G)	destination of sensor G

Figure 6.1: Notation guide to state transition diagrams.

6.3.1 Tram Detection

The control system uses the control counters to detect where in the network trams are located. For each sensor in the network the control system has a related control counter. Each time a sensor becomes *high* the related control counter is increased by one. The difference between two control counters related to two sensors on a linear rail with no points in between, expresses the number of trams on the track between the two sensors. For points the calculations depends on the actual point construction.

Thus, by placing sensors in the network it is possible to maintain precise information of where trams are located in the network.

6.3.2 The Route Dispatcher STS

The route dispatcher STS has one state, called *ready to allocate*. This state has a transition rule for each route:

1-1 **Conditions:** The route is requested, not reserved, and no conflicting routes are reserved. Furthermore, the entry signals for safety conflicting routes are set to *stop*, this last condition is an extra safety precaution and redundant as it is also checked that these routes are not reserved.
Actions: The route becomes reserved and is no longer requested.

As the route dispatcher STS has more than one order of execution, a priority of the transition rules must be introduced. This priority corresponds to the order of routes explained in section 3.3.2.

The route dispatcher STS for the running example can be presented by the state transition diagram in figure 6.2. Please note that only the transition rule for route 1 is specified in the figure.

6.3.3 The Route Controller STSs

Each route controller STS has four different states:

1. The route is not allocated.
2. The points along the route are being set.

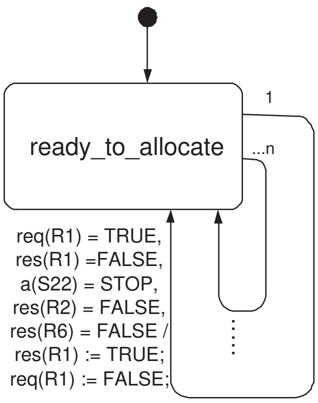

Figure 6.2: State transition diagram for Route Dispatcher.

3. The route is allocated but not yet in use.
4. The route is allocated and in use.

Each state has a transition rule, causing a state change to the following state (i.e. from state 1 to 2 and so forth).

1-2 **Conditions:** The route is requested, and the entry signals of safety conflicting routes are set to *stop*. The last condition is an extra safety precaution and redundant as these routes are not reserved.
Actions: The points on the route are requested to be set to the values given by the point position table.

2-3 **Conditions:** The points on the route are set as requested in rule *1-2*.
Actions: The entry signal of the route is requested to be set to the value given by the signal setting table.

3-4 **Conditions:** The entry signal is set as requested in rule *2-3*, and the control counter corresponding to the first sensor of the route is larger than zero, meaning the sensor has been passed.
Actions: The entry signal is requested to be set to *stop*.

4-1 **Conditions:** The entry signal is set to *stop*, and the control counters of the first and the last sensor on the route are equal, meaning that the route has been finished.
Actions: All control counters corresponding to sensors on the route are reset (set to zero), and the route is no longer reserved.

The STS for the route controller of route 1 is given by a state transition diagram in figure 6.3.

6.3.4 The Counter Controller STSs

For each control counter there is a controller that increases the value of the control counter by one, when a tram passes the sensor. Thus, a control counter can be in one of two states:

1. Ready to count.
2. Counted.

The STS has two transition rules:

1-2 **Conditions:** The related sensor is *high*, meaning that a tram passes the sensor.

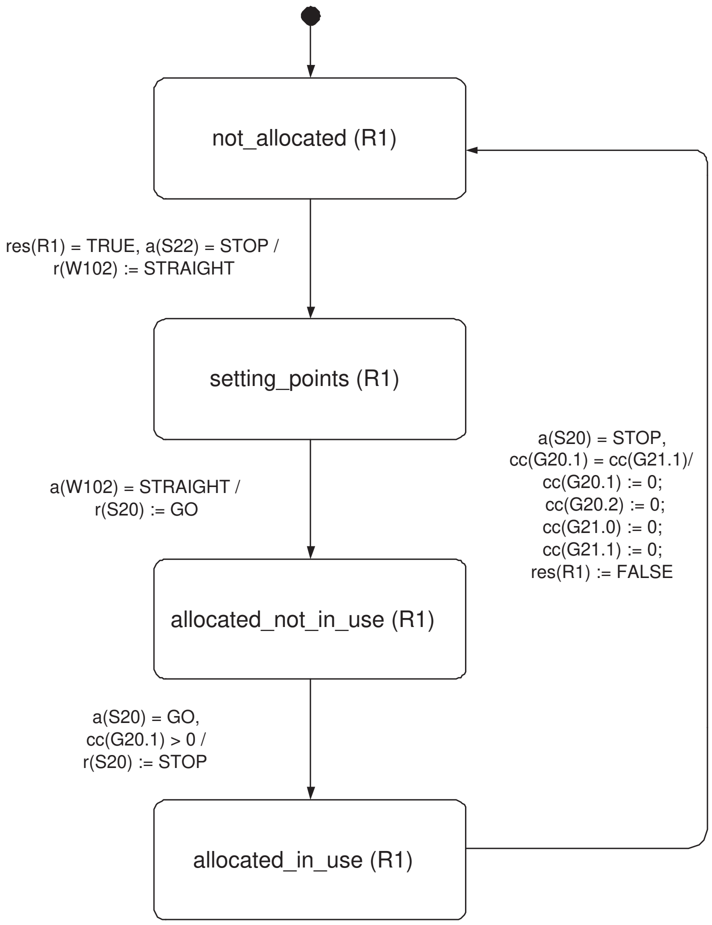

Figure 6.3: State transition diagram for Route Controller for route 1.

Actions: The control counter is increased by one.

2-1 **Conditions:** The related sensor is *low*.

Actions: No variables are modified except the control mode variable.

A counter controller for the control counter of a sensor *G1* is presented by the state transition diagram in figure 6.4.

6.4 STSs Simulating the Physical Environment

The STS simulating the physical environment of the network is a parallel combination of STSs simulating the behavior of each device.

$$NTS = (sensor_1 \parallel ... \parallel sensor_n \parallel signal_1 \parallel ... \parallel signal_i \parallel point_1 \parallel ... \parallel point_j)$$

Where n, i, and j are the number of sensors, signals, and controllable points, respectively.

6.4.1 STSs Simulating Signals

Each signal in a network is simulated by an STS. The purpose of the STS is to update the actual state of the signal according to requests from the control system given by the signal control variable.

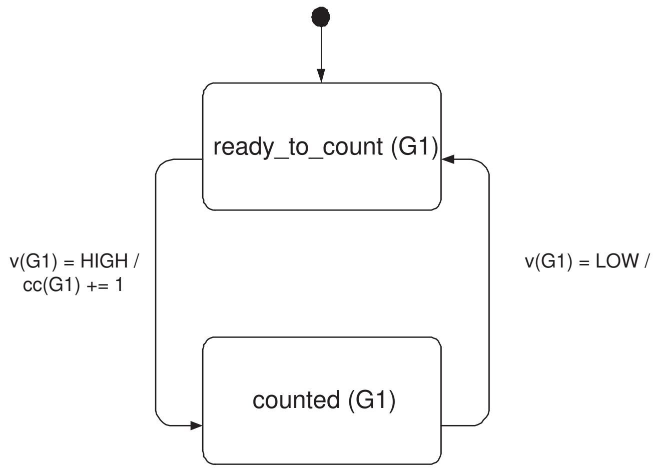

Figure 6.4: State transition diagram for Counter Controller for sensor G1.

Each such STS has one state:

1. Working signal

With one transition:

1-1 **Conditions:** The requested signal state is different from the actual state.
Actions: The actual signal state is set to the requested signal state.

This is the simplest possible way to simulate the signal, and it assumes that the shifting of signals is almost instantaneous and that the signals do not fail in shifting.

A generic signal STS is presented by the state transition diagram in figure 6.5.

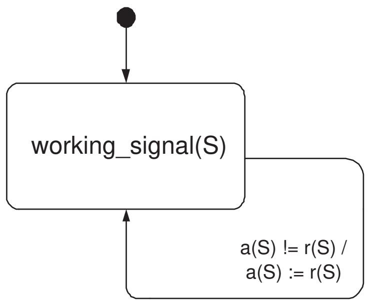

Figure 6.5: Generic state transition diagram simulating signal *S*.

6.4.2 STSs Simulating Points

Each controllable point in the network is simulated by an STS. The purpose of the STS is similar to the one for signals, being updating the actual state on the request of the control system. The main difference is the timing of the devices. While signals are assumed to switch instantly from one state to another, points have an intermediate state because the switching is not instantaneous.

Thus, if a point is requested to switch it will start switching. If no other request is issued while switching, it will complete the process. Otherwise, if the point is requested to be in another position it will start switching to that position.

Hence, the points simulated cannot fail, but nevertheless it is possible to issue a command to a point while switching, effectively reversing the direction of the switching.

Thus, the generic point has one state:

1. Working point

Six transition rules are necessary to model the switching of the point:

1-1 **Conditions:** Point is set to *straight* and requested to be set to *turn*.
Actions: Point is set to *move_ to_ turn*, meaning the point is in the process of switching to turn position.

1-1 **Conditions:** Point is set to *move_ to_ turn* (in the process of switching) and requested to be set to *turn*.
Actions: Point is set to *turn*, meaning the point has finished the process of switching position.

1-1 **Conditions:** Point is set to *move_ to_ turn* (in the process of switching) and requested to be set to *straight*.
Actions: Point is set to *move_ to_ straight*, meaning the point has changed direction of the switching.

1-1 ...

The rest of the rules are similar to the ones above, meaning that all instances of *straight* should be replaced with *turn*, and vica versa in the names of the positions. E. g. *move_ to_ turn* becomes *move_ to_ straight*.

A generic point STS simulating point P is presented by a state transition diagram in figure 6.6.

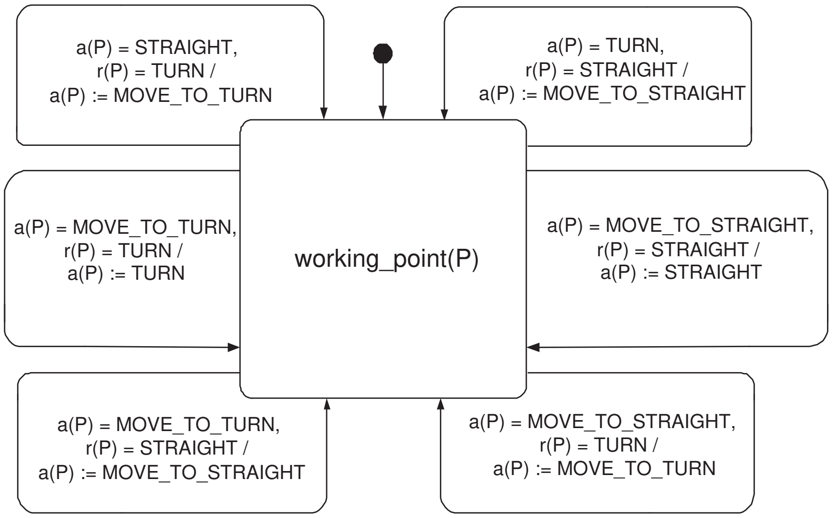

Figure 6.6: Generic state transition diagram simulating point P

6.4.3 STSs Simulating Sensors

The STSs simulating the sensors implicitly express that trams are assumed to drive according to the rules for driving in tramway networks. This includes no trams passing signals showing *stop* and no trams passing a sensor in the middle of a network without having passed any of the neighbouring sensors.

There are four different types of sensors defined by their position in the network. Each type has a distinct construction of STS simulating the sensor.

The types are the entry border sensors (just before the start sensor of a route), the exit border sensors (the end sensor of a route), the point sensors (stem, turn, or straight sensor for at least one point), and the internal sensors (all other sensors).

A *neighbour* for a sensor is another sensor which together with the considered sensor defines a segment. Thus, a border sensor has one neighbour, a sensor at the stem of a point has three neighbours, and the rest of the sensors have two neighbours. To simulate the sensors, counters and destinations are used. The sensor counter is used for modelling when a sensor can become *high* using flow calculations, and the destination is used for describing certain safety properties of the network (see chapter 10).

All sensor STSs have two states:

1. Sensor low
2. Sensor high

The number of transition rules of the STS depends on the type of the sensor, but all have the following two standard rules:

1-1 *Updating the destination of the sensor*[3].
Conditions: The destination must be different from ϵ (non-empty), and the counter of the sensor must be less than or equal to the counters of neighbouring sensors.
Actions: The destination is set to ϵ.

2-1 *Sensor becomes low.*
Conditions: The sensor is *high*.
Actions: The sensor is set to *low*.

In addition to this each sensor has rules for deciding when the sensor can become *high*, based on the flow.

The updating of destination has the highest priority, rules for when the sensor can become high have next highest priority, and rules for going from high to low have lowest priority.

Flow

The flow in a direction into a sensor is basically the number of trams driving towards the sensor from the considered direction, and the flow out of a sensor is the number of tram driving away from the sensor in that direction. In figure 6.7 the flow of the internal sensor B is illustrated and the calculations are given in figure 6.8. These are quite obvious, as B only has two neighbouring sensors.

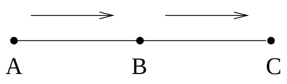

Figure 6.7: Flow of internal sensor.

The flow in a point sensor is slightly more complicated, as it has more than two neighbours. The flow of a point sensor B in one direction is illustrated in figure 6.9, and the calculation for both directions is presented in figure 6.10

[3]Due to shortcomings of the tool used to generate the state transition diagrams ϵ is depicted as E.

direction	flow into B	flow out of B
→	c(A) - c(B)	c(B) - c(C)
←	c(C) - c(B)	c(B) - c(A)

Figure 6.8: Flow calculation for internal sensor.

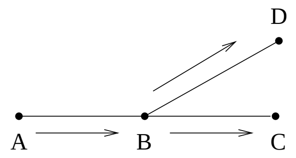

Figure 6.9: Flow of point sensor.

direction	flow into B	flow out of B
→	c(A) - c(B)	c(B) - (c(C) + c(D))
←	(c(C) + c(D)) - c(B)	c(B) - c(A)

Figure 6.10: Flow calculation for point sensor.

The flow calculations of point sensors can be a little more complicated depending on the specific constructions, but follow the basic principle presented here.

Entry Border Sensors

The transition rule for going from *low* to *high*, for an entry border sensor STS, is presented:

1-2 **Conditions:** The sensor is *low*. The sensor can always become *high* as there are no restriction on the traffic entering the network at this point.
Actions: The sensor is set to *high*, the counter for the sensor is increased by one, and the destination of the sensor is set to the neighbouring sensor, as the tram is driving into the network.

An STS for an entry border sensor *G20.0* from the running example is presented by a state transition diagram in figure 6.11.

Exit Border Sensors.

The transition rule which is unique for STSs simulating exit border sensors is described:

1-2 **Conditions:** The sensor is *low*, and the counter is less than the counter of the neighbouring sensor, being the sensor the tram is coming from; thus, the flow into the sensor is larger than zero.
Actions: The sensor is set to *high*, the counter is increased by one, and the destination is set to ϵ as the tram is leaving the network.

An STS for the exit border sensor *G21.1* from the running example is presented by a state transition diagram in figure 6.12.

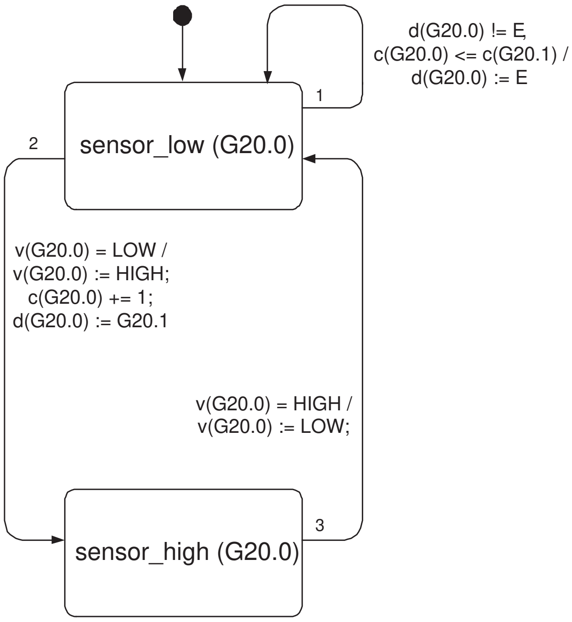

Figure 6.11: State transition diagram simulating the entry border sensor *G20.0*.

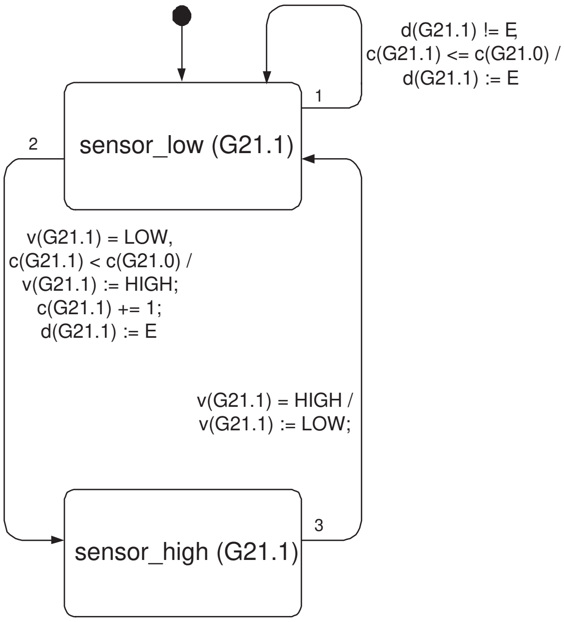

Figure 6.12: State transition diagram simulating the exit border sensor *G21.1*.

Internal Sensor

The transition rules which are unique for STSs simulating internal sensors are described. The tram can arrive from two sensors leading to two rules:

1-2 **Conditions:** The sensor is *low*, and the counter flow into the sensor from the first neighbouring sensor is larger than zero. Furthermore, if a signal is located at the sensor and visible from the first neighbouring sensor, it shows *go*.
Actions: The sensor is set to *high*, the counter is increased by one, and the destination is set to the second neighbouring sensor.

1-2 **Conditions:** The sensor is *low*, and the counter flow into the sensor from the second neighbouring sensor is larger than zero. Furthermore, if a signal is located at the sensor and visible from the second neighbouring sensor, it shows *go*.
Actions: The sensor is set to *high*, the counter is increased by one, and the destination is set to the first neighbouring sensor.

Please note that the notion of the first and second neighbouring sensor used above only serves to discern between the two different sensors in the network a tram can arrive from, as no order is enforced. Additionally, a signal is only visible from the direction of a sensor if that sensor is a border sensor.

An STS for an internal sensor *G1* is presented by a state transition diagram in figure 6.13. Please note that the running example has no internal sensors, and therefore the diagram is for a non-specific sensor, with the neighbouring sensors *G0* and *G2*.

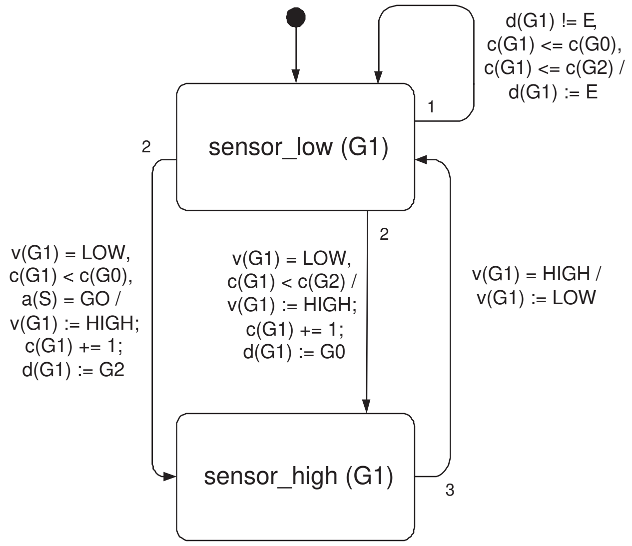

Figure 6.13: State transition diagram simulating an internal sensor *G1* with a signal, visible from *G0*.

Point Sensor

Point sensors have the most complicated STSs, as the specific rules and conditions are dependent on the exact construction of the point and the surroundings, and on whether the point is controllable or not.

As for the other sensor types there is a rule for each sensor the tram can arrive from. The specific condition saying when a tram can arrive from a direction depends on the flow. Thus, for a specific sensor to be *high* the flow into the sensor must be larger than zero in one of the directions. Because points can be connected in several ways there are several different flow calculations.

Generically the sensor STS for the stem and either branch of a point has a rule for each valid combination of entering and leaving the point:

1-2 **Conditions:** The sensor is *low*, and the flow into the sensor from the considered direction is larger than zero, making it possible for a tram to pass the sensor

from the considered direction. If a signal is visible from the direction it shows *go*. If the involved points are controllable the setting is correct.
Actions: The sensor is set to *high*, the counter is increased by one, and the destination is set to be the first sensor in the direction the tram is leaving. Please note that this is dependent on the setting of the point if it is controllable.

The STS can be generated from the properties of which points have the sensor as stem, turn, or straight sensor. Thus, according to 2.4.2 the following constructions are possible:

- The sensor is stem sensor for a point and ...
  The sensor is straight sensor for another point.
  The sensor is turn sensor for another point.
  The sensor is not turn or straight for another point.
- the sensor is straight sensor for a point and ...
  The sensor is also straight sensor for another point.
  The sensor is turn sensor for another point.
  The sensor is not turn or straight for another point.
- the sensor is turn sensor for a point and ...
  The sensor is also turn sensor for another point.
  The sensor is not turn or straight for another point.

Therefore, the entire construct will not be presented. Instead, one of the constructions of the above will be presented to illustrate the flow condition construction for point sensor STSs.

The STS for a point sensor that is the stem of a point and not in any way related to any other point has three transition rules in addition to the standard rules for sensor STSs:

1-2 *The tram arrives from stem side, and the point is set to straight.*
Conditions: The sensor is *low*, the counter flow into the sensor from stem side is larger than zero. Furthermore, the point is in *straight* position (this condition is only valid for controllable points), and if there is a signal at the sensor visible from stem side it shows *go*.
Actions: The sensor is set to *high*, the counter is increased by one, and the destination is set to the straight sensor of the point.

1-2 *The tram arrives from stem side, and the point is set to turn. This rule is only valid if the point is controllable because otherwise the setting is static corresponding to straight.*
Conditions: The sensor is *low*, and the flow into the sensor from stem side is larger than zero. Furthermore, the point is in *turn* position, and if there is a signal at the sensor visible from the stem side it shows *go*.
Actions: The sensor is set to *high*, the counter is increased by one, and the destination is set to the turn sensor of the point.

1-2 *The tram arrives from either branch.*
Conditions: The sensor is *low* and the flow into the sensor from branch side is larger than zero. This flow is calculated based on the sum of counters for the straight and turn sensor minus the counter for the stem sensor as illustrated in figure 6.9.
Actions: The sensor is set to *high*, the counter is increased by one, and the destination is set to be the neighbouring sensor that is neither turn or straight branch for the point.

An STS for a point sensor G20.1 is illustrated by a state transition diagram in figure 6.14.

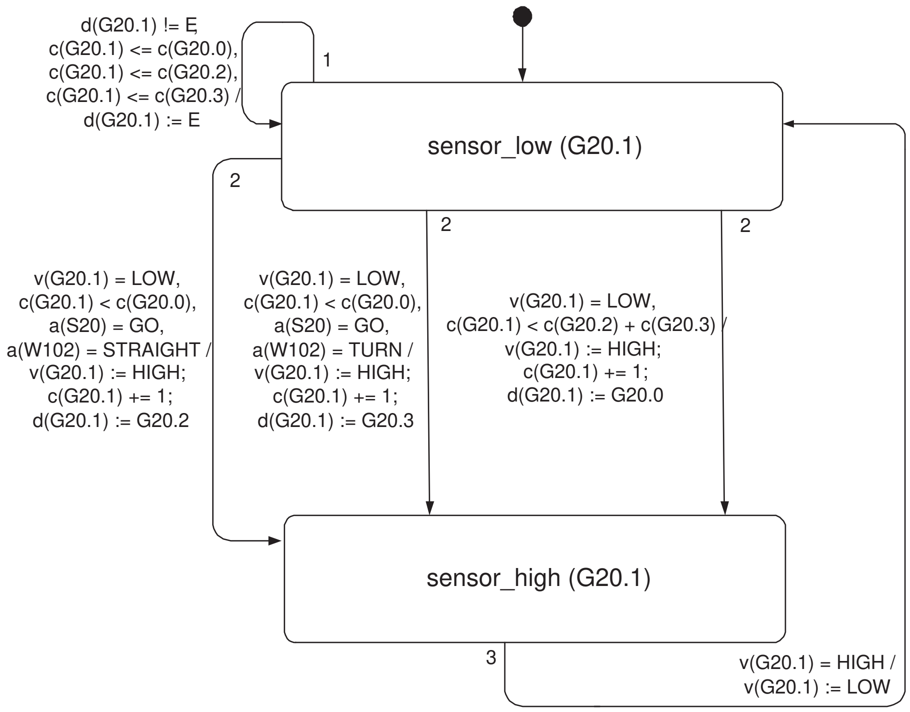

Figure 6.14: State transition diagram simulating point sensor G20.1.

6.4.4 Interpreter

As presented in section 6.3 the STS control components should be executed as a parallel combination. This parallel combination can be refined by introducing a control algorithm, executing the components in order, and only firing one rule per STS at a time. Thereby, only a specific subset of all possible executions of the parallel combinations are considered. The executions that are left out with this refinement are not necessary to ensure the correct behaviour of the control system, and furthermore the model is simpler than an actual parallel execution of the components.

The specific order of execution of the control components as chosen in this thesis is:

1. Counter Controllers.
2. Route Dispatcher.
3. Route Controllers.

This order is chosen to complement the natural order of the system, as the entry border sensor is first passed, then the route is requested, then allocated, and then driven.

The order of the counter controllers corresponds to the order of the corresponding sensors in the sensorids list in the model of the DST-L. The order of the route controllers corresponds to the priority of the corresponding routes.

6.5 Chapter Summary

In summary the different control components and a simulation of the physical environment have been given an STS semantics, and the specific STSs have been described informally and more formally by state transition diagrams. Furthermore, a control algorithm executing the control components in order has been introduced.

Chapter 7

Model of STS Language

In this chapter an STS language (STS-L) enabling the representation of control components and physical environment as presented in chapter 6 is modelled in RSL. First, the approach followed when modelling is presented in section 7.1, and then each of the language elements: abstract syntax (section 7.2), static semantics (section 7.3), and operational semantics (section 7.5) are presented.

For a short introduction to the syntax of STSs please refer to index A

7.1 Model Approach

When modelling the STS-L used to describe the physical environment and control system there are several possible approaches.

The STS-L model consists of a model of the sentences in the abstract syntax, which sentences are well-formed in the static semantics, and the meaning of the sentences in the operational semantics.

In this context the models can be considered a formal specification of the STSs presented in appendix A.

Furthermore, the global state space is included in the model, functioning as the environment for the STS-L, as the values of objects are kept here.

The conditions and actions concern objects. These objects can in the general case have any type. It is possible to model a generic STS-L, not dependent on the typing of objects in the conditions and actions, but it is also possible to model the language in a more domain specific way.

The question is how to store information about objects to be used when evaluating conditions and processing actions. I.e. how the information in the global state space is distributed.

In the general case all types of objects are considered to be of one uniform object type, with added information about whether the object is a sensor, signal, and so forth.

Generalising the information needed for each object would lead to a generic state space easily adaptable to other domains. The problem with this is to actually construct a satisfactory generalisation of the objects.

When considering the control system and physical environment of the tramway domain, several different types of objects exist. Physical objects for signals, sensors, points, and control variables representing requested values of such objects exist in the control system. Furthermore, the control system has additional control variables to help in the processing of data. The problem is that the information associated with each type of object varies. A sensor has information about a counter, current destination, and the state of the sensor. For a signal both a physical value (actual value) and a requested value exist in the control system. Hence, a generalisation of objects is troublesome.

Therefore, constructing a more domain specific STS-L, modelling both state space and sentences to suit the needs for representing the objects of the tramway domain is another possibility. The problem with this modelling approach is that the model by being more specific possibly also becomes more complicated.

As opposed to the prototype implementation presented in [Pel01], a domain specific approach as described above has been chosen. This is mainly to create a more direct connection between the domain specific language and the STS-L used, and because a concise, precise, and elegant generalisation of objects is not readily available.

This also implies that the STS-L constructed is minimal in the sense that only aspects needed to model the control components and the physical environment have been modelled. Hence, certain aspects of the general STS-L presented in appendix A are left out.

The modelling of the abstract syntax, static semantics, and operational semantics are described in the following sections. The entire specification can be found in appendix B.3.

7.2 Abstract Syntax

The abstract syntax is formulated in the RSL scheme STS_abstract_syntax found in appendix B.3.1.

Identifiers are introduced for the devices and for control variables. They are specified using a super type STSLId similarly to the one used in the modelling of the DSTL:

```
type
  STSLId,
  STSLSensorId = {| id : STSLId • id ∈ elems STSLsensorids |},
  STSLPointId = {| id : STSLId • id ∈ elems STSLpointids |},
  STSLSignalId = {| id : STSLId • id ∈ elems STSLsignalids |},
  CMVId = {| i : STSLId • i ∈ elems cmvids |},
  CCountId = {| i : STSLId • i ∈ elems ccountids |},
  RCVId = {| i : STSLId • i ∈ elems rcvids |}

value
  STSLsensorids : STSLId*,
  STSLpointids : STSLId*,
  STSLsignalids : STSLId*,
  cmvids : STSLId*,
  ccountids : STSLId*,
  rcvids : STSLId*
```

Axioms, called [no_duplicates_in_STSLsensorids] and so forth, are introduced ensuring that no duplicates exist in the lists. Furthermore, an axiom [distinct_stsl_ids] is introduced ensuring that the STSLId-lists do not have common elements.

An STS consists of a number of rules, and is therefore specified as a list of rules:

type Sts = Rule*

The ordering of the rules in the list models the priority of the different rules, such that the first rule in the list has highest priority, and so forth.

A rule consists of a number of conditions which must evaluate to true for the rule to be fired. A rule is fired by processing a number of actions. A rule in the language is therefore formulated as a list of conditions and a list of actions:

type Rule = Condition* × Action*

A condition formulates a necessary relation between objects and object states which must be fulfilled for a rule to be fired. The relation is between an object and an expression. The expression is either another object, a state, or an expression of objects.

This means that a condition consists of three parts, a pointer to an object in the state space (IdComp), a relation, and an expression.

type Condition :: idcomp_of : IdComp relation : Relation expr : Expr

An action assigns new values to objects and is modelled similarly to conditions with Operation instead of Relation:

type Action :: idcomp_of : IdComp operation : Operation expr : Expr

A variant type IdComp is introduced, pointing to a specific position of an object in the global state space:

type
IdComp ==
mk_actualSen(senid : STSLSensorId) |
mk_counterSen(senid2 : STSLSensorId) |
mk_destinationSen(senid3 : STSLSensorId) |
...

The table in figure 7.1 relates the notations used in the state transition diagrams (STD) in chapter 6 to the corresponding IdComps.

IdComp	STD Notation
mk_actualSen(G)	v(G)
mk_counterSen(G)	c(G)
mk_destinationSen(G)	d(G)
mk_actualP(W)	a(W)
mk_requestP(W)	r(W)
mk_actualSig(S)	a(S)
mk_requestSig(S)	r(S)
mk_cmv(cmvid)	-
mk_ccount(G)	cc(G)
mk_routerequest(R)	req(R)
mk_routereserved(R)	res(R)

Figure 7.1: Relation between the STD notations and the IdComps.

The expression of both conditions and actions is modelled by a variant type Expr, consisting of an IdComp part if it is a pointer to an object, a State if it is an object state, and an AddIdComps if it is a more complicated expression.

```
type
  Expr ==
    mk_IdComp(idco : IdComp) |
    mk_State(st : State) |
    mk_AddIdComps(aics : AddIdComps),

  State ==
    mk_SensorState(sensorstate : SensorState) |
    mk_PointState(pointstate : PointState) |
    mk_SignalState(signalstate : SignalState) |
    mk_CMVState(cmvstate : CMVState) |
    mk_RCVState(rcvstate : RCVState) |
    mk_Counter(count : Counter) |
    mk_Destination(dest : Destination)
```

The AddIdComps is basically an expression that can be further divided into two expressions with an infix operator[1]. The model is minimal in the sense that since only the addition of the values of two IdComp is needed, the AddIdComps is a cartesian product of two IdComp, and it is given in the semantics of such an expression that an addition of the values must be performed.

The relations which have been found sufficient to express the necessary conditions are given by a type Relation. Only the relations needed to represent the conditions used in the control components and the physical environment have been modelled. The table in figure 7.2 explains the relations used in the model.

Relations		
Id	**Name**	**Description**
REL_EQUAL	Equal	The equality relation (=)
REL_NOT_EQUAL	Not Equal	The not equal relation (≠)
REL_GT	Greater than	The greater than relation (>)
REL_LT	Less than	The less than relation (<)
REL_LE	Less than or equal	The less than or equal relation (≤)

Figure 7.2: Relations used in the model.

```
type
  Relation == REL_EQUAL | REL_NOT_EQUAL | REL_GT | REL_LT | REL_LE
```

The operations which have been found sufficient to express the necessary actions are given by the type Operation. The table in figure 7.3 explains the operations used in the model.

Operations		
Id	**Name**	**Description**
OP_SET	Assignment	Assigns an object state to an object (:=).
OP_PLUS_SET	Add constant	Adds a natural number to the object state of a counter (+=).

Figure 7.3: Operations used in the model.

```
type
  Operation == OP_SET | OP_PLUS_SET
```

[1]Not to be confused with the operation of an action.

A variant type is associated to each type of object in the model. The variant types represent the possible states of objects of the different types. For example the possible states of a sensor are S_HIGH and S_LOW given by the variant type SensorState. The states of the other types of objects are defined the same way (see appendix B.3.1).

```
type
  SensorState == S_HIGH | S_LOW
```

7.3 Static Semantics

The static semantics constitutes well-formedness predicates for the sentences in the language. The static semantics are specified in the RSL scheme STS_static_semantics which can be found in appendix B.3.2.

To help in the formulation of well-formedness predicates for the STS language a number of different type_of functions has been declared, each returning the type of a sentence of the STS language.

An STS is well-formed if each rule in the STS is well-formed. A rule is well-formed if the condition list and action list of the rule are well-formed. A condition list and an action list are well-formed if all conditions and actions, respectively, are well-formed.

A condition is well-formed if the types of the IdComp and the Expr parts are the same. Furthermore, if the relation is a "greater than" relation, "less than" relation, or "less than or equal" relation, the type of the IdComp and the Expr parts must both be COUNTER. This is because the COUNTER is of type Nat and the semantics of these relations is well defined within the domain of natural numbers[2]:

```
value
  wf_cond : Condition → Bool
  wf_cond(cond) ≡
   type_of(idcomp_of(cond)) = type_of(expr(cond)) ∧
   (relation(cond) = REL_GT ∨ relation(cond) = REL_LE
   ∨ relation(cond) = REL_LT ⇒ type_of(idcomp_of(cond)) = COUNTER )
```

An action is well-formed if the types of the IdComp and the Expr parts are identical. Furthermore, if the operation is OP_PLUS_SET, type of the IdComp and the Expr parts must both be COUNTER. Similarly to conditions this is because the state of a counter is of type Nat:

```
value
  wf_act : Action → Bool
  wf_act(act) ≡
    type_of(idcomp_of(act)) = type_of(expr(act)) ∧
    (operation(act) = OP_PLUS_SET ⇒ type_of(idcomp_of(act)) = COUNTER)
```

7.4 State Space

The state space is specified as part of the operational semantics in the RSL scheme STS_operational_semantics found in appendix B.3.3.

[2]Also integers, but natural numbers are better suited for representing counters in the model.

The state space contains information about the values of all objects. The type StateSpace is a short record type with mappings from object identifiers to the corresponding object states. The distribution of information is according to the different object types:

```
type
  StateSpace ::
    sensoraux : SensorAux
    sensorstates : SensorStates
    pointstates : PointStates
    signalstates : SignalStates
    cmvstates : CMVStates
    ccountstates : CCountStates
    rcvstates : RCVStates
```

All the maps must be total, specified by enforcing that the domain of each map consists of all the identifiers of the relevant type. Each of the mappings in the state space are explained below:

- sensoraux maps each SensorId to the counter and destination of the sensor:

  ```
  type
    SensorAux =
      {| m : STSLSensorId →m (Counter × Destination) • dom m = { x | x : STSLSensorId } |}
  ```

- sensorstates maps each STSLSensorId to the current state of the sensor.
- pointstates maps each STSLPointId to a tuple containing the actual and requested state of the point.
- signalstates maps each STSLSignalId to a tuple containing the actual and requested state of the signal.
- cmvstates maps each CMVId to the current state of the control mode variable (cmv). The cmv represents the current state of the associated STS.
- ccountstates maps each CCountId to the current value of the control counter.
- rcvstates maps each RCVId to a tuple of rcv states, one representing whether the corresponding route is requested and one whether the route is reserved.

7.5 Operational Semantics

An interpreter function expresses the operational semantics of the STS language. The operational semantics is specified in the RSL scheme STS_operational_semantics found in appendix B.3.3.

The Interpreter The basic concepts of the interpreter used in the STS-L is presented in section 6.4.4. The interpreter only maintains the order in which the rules are evaluated. For a chosen rule it calls another function performing the actual interpretation of the rule.

The operational semantics is therefore stated by an interpreter function which for a given STS evaluates the conditions of the rules in order and, if they evaluate to true, processes the actions of that rule. When one rule has been processed the updated state space is returned and the interpreter is terminated.

```
value
  interpret_sts : Sts ~→ StateSpace → StateSpace
```

```
interpret_sts(sts)(ss)≡
  if sts = ⟨⟩ then
    ss
  else
    let (result, fired) = interpret_rule(hd sts)(ss) in
      if fired then result else interpret_sts(tl sts)(ss) end
    end
  end
pre wf_sts(sts)
```

Each rule is processed by the function interpret_rule. If the actions of the rule are processed, the function returns the updated state space and true, indicating a change in the state space. Otherwise, the unchanged state space and false is returned.

```
value
  interpret_rule : Rule ~→ StateSpace → StateSpace × Bool
  interpret_rule(cl, al)(ss)≡
    if eval_cond_list(cl)(ss) then
      (process_action_list(al)(ss), true)
    else
      (ss, false)
    end
  pre wf_rule(cl, al)
```

The evaluation of a condition is based on the type of the relation, but basically it is applying the relation to the states resulting from applying eval_idcomp, to the idcomp part, and eval_expr to the expr part. This eval_idcomp retrieves the value from the state space by calling the corresponding retrieval functions for instance get_actual_point_state. eval_expr is slightly more complicated, checking whether the expression is an IdComp, in which case it calls eval_idcomp, an object state, in which case it returns the state, or an AddIdComps, in which case it calls eval_addidcomps, which in turn calls eval_idcomp on both IdComps and adds them.

```
value
  eval_cond : Condition ~→ StateSpace → Bool
  eval_cond(cond)(ss)≡
    let s1 = eval_idcomp(idcomp_of(cond))(ss), s2 = eval_expr(expr(cond))(ss) in
      case relation(cond) of
        REL_LE → count(s1) ≤ count(s2),
        REL_GT → count(s1) > count(s2),
        REL_LT → count(s1) < count(s2),
        REL_EQUAL → s1 = s2,
        REL_NOT_EQUAL → s1 ≠ s2
      end
    end
  pre wf_cond(cond)
```

The processing of an action is based on the operation, and the state space is updated accordingly. The function process_action uses nested case expressions, deciding which part of the state space should be updated and how, and then calling the correct update function such as set_actual_point_state. The update function updates the specific part of the state space according to the operation and the new state. Please refer to appendix B.3.3 for the update functions.

```
value
```

```
process_action : Action ~→ StateSpace → StateSpace
process_action(act)(ss) ≡
  let new_state = eval_expr(expr(act))(ss) in
    case operation(act) of
      OP_PLUS_SET →
        case idcomp_of(act) of
          mk_counterSen(sd) → plus_set_physical_counter_state(sd, count(new_state))(ss),
          ...
        end,
      OP_SET →
        case idcomp_of(act) of
          mk_actualSen(sd) → set_actual_sensor_state(sd, sensorstate(new_state))(ss),
          mk_destinationSen(sd) → set_sensor_destination(sd, dest(new_state))(ss),
          ...
        end
    end
  end
pre wf_act(act)
```

An example of a retrieval function is get_control_counter_state, which retrieves the value of the control counter corresponding to the identifier in the ccountstates map of the state space. Likewise, set_control_counter is an update function that sets the new value of the control counter by overriding the ccountstates map. All retrieval and update functions are specified similarly, the difference being the maps of the state space consulted and exactly which values are updated/retrieved.

```
value
  get_control_counter_state : CCountId → StateSpace → Counter
  get_control_counter_state(ccd)(ss) ≡ ccountstates(ss)(ccd),

  set_control_counter_state : CCountId × Counter → StateSpace → StateSpace
  set_control_counter_state(cid, new_state)(ss) ≡
    mk_StateSpace(
      sensoraux(ss),
      sensorstates(ss),
      pointstates(ss),
      signalstates(ss),
      cmvstates(ss),
      ccountstates(ss) † [cid ↦ new_state],
      rcvstates(ss) )
```

7.6 Chapter Summary

The model of the STS-L corresponding to the STS syntax presented in appendix A has been introduced. This includes the abstract syntax, static semantics, and the operational semantics.

Chapter 8

Model of Generic Control Components and Physical Environment

In this chapter a model of the generic control components and the physical environment introduced in chapter 6, and an initial state space are presented.

In section 8.1 a specification of the initial values in the state space is presented, in section 8.2 the control components are presented, and in section 8.3 the physical environment is presented.

The STS-L syntax is accessible through the global object S.

The entire model is in the RSL scheme GENERATOR found in appendix B.5.

8.1 Initial State Space

The state space is modelled in section 7.4 as a number of mappings from object identifiers to object data as explained in section 6.1. Only the identifiers, and not their initial values are dependent on a concrete network in the initial state space. This means that a concrete object type always has a specific initial value.

The initial values for each object are explained below:

- initial_sensor_aux: For each STSLSensorId the counter initial value is *0* and the destination is ϵ.
- initial_sensor_states: For each STSLSensorId the initial value is S_LOW.
- initial_point_states: For each STSLPointId both the initial actual and requested state are S_STRAIGHT.
- initial_signal_states: For each STSLSignalId both the initial actual and requested state are S_STOP:

```
initial_signal_states : S.SignalStates =
  [ signalid_to_stslsignalid(N.signalids(i)) ↦ (S.S_STOP, S.S_STOP)
    | i : Nat • 0 < 1 ∧ i ≤ len N.signalids ]
```

- initial_cmv_states:

- For the control mode variable (CMVId) of the route dispatcher, the initial value is READY_TO_ALLOCATE.
- For the CMVId of each RCVId, corresponding to the route controllers the initial value is NOT_ALLOCATED.
- For the CMVId of each SignalId and PointId, the initial value is WORKING_DEVICE. This is the initial state of the STS simulating the device. Please note that the *working_signal* and *working_point* values presented in chapter 6, are abstracted in the model by WORKING_DEVICE.
- For the CMVId of each SensorId, the initial value is SENSOR_LOW. This is the initial state of the STS simulating the sensor.
- For the CMVId of each control counter (CCountId), the initial value is READY_TO_COUNT. This is the initial state of the STS representing the control counter.

- initial_ccount_states: For each CCountId the initial value is *0*.
- initial_rcv_states: For each route control variable (RCVId) the two initial values are false. Thus, no routes are initially requested or reserved.

8.2 STSs Representing Control Components

The STSs representing the control components are the route_dispatcher STS, the route_controllers STSs and the counter_controllers STSs. The construction of these depends on the concrete network and interlocking tables. Therefore, it is performed based on the description formulated in the DST-L.

As stated in chapter 7, an STS is a list of rules, and a rule is a list of conditions and a list of actions. Thus, the basic building blocks for an STS are the conditions and actions. A condition is constructed as:

```
mk_Condition(IdComp, Relation, Expr)
```

where IdComp, Relation, and Expr is presented in chapter 7. An action is constructed as:

```
mk_Action(IdComp, Operation, Expr)
```

where Operation is also presented in chapter 7.

The Route Dispatcher The route dispather is constructed corresponding to the description given in section 6.3.2 having one transition rule for each route in the network.

The Route Controller For each route in the network a route controller is constructed. This generation corresponds to the description given in section 6.3.3.

The Counter Controller For each sensor in the concrete network a counter controller is constructed, since a control counter exists for each sensor. This corresponds to the description given in section 6.3.4. The construction of a single rule, describing the transition from "ready to count" state to the "counted" state for the sensor identifier the i, is presented here:

```
⟨
  S.mk_Condition
    (
      S.mk_cmv(ccountid_to_cmvid(S.ccountids(i))),
      S.REL_EQUAL,
      S.mk_State(S.mk_CMVState(S.READY_TO_COUNT))
    )
⟩ ⌢
⟨
  S.mk_Condition
    (
      S.mk_actualSen(sensorid_to_stslsensorid(N.sensorids(i))),
      S.REL_EQUAL,
      S.mk_State(S.mk_SensorState(S.S_HIGH))
    )
⟩,
⟨
  S.mk_Action
    (
      S.mk_ccount(S.ccountids(i)),
      S.OP_PLUS_SET,
      S.mk_State(S.mk_Counter(1))
    )
⟩ ⌢
⟨
  S.mk_Action
    (
      S.mk_cmv(ccountid_to_cmvid(S.ccountids(i))),
      S.OP_SET,
      S.mk_State(S.mk_CMVState(S.COUNTED))
    )
⟩
```

8.3 STSs Simulating the Physical Environment

The STSs simulating the physical environment are the sensor_simul, the signal_simul, and the point_simul. The construction is performed based on the network description formulated in the DST-L.

STSs Simulating Signals For each signal in the concrete network an STS is constructed, simulating the state changes as explained in section 6.4.1.

An example of a condition for a signal is:

```
S.mk_Condition
  (
    S.mk_cmv(signalid_to_cmvid(N.signalids(i))),
    S.REL_EQUAL,
    S.mk_State(S.mk_CMVState(S.WORKING_DEVICE))
  )
```

where the first part expresses that it is the control mode variable for the signal in question being examined. The rest of the condition merely states that the value of this control mode variable must be WORKING_DEVICE.

STSs Simulating Points For each controllable point in the concrete network an STS is constructed, simulating the state changes as explained in section 6.4.2.

STSs Simulating Sensors For each sensor in the concrete network an STS is constructed, simulating the state changes of the sensors as explained in section 6.4.3. The STS depends on whether the sensor is an entry border sensor, an exit border sensor, an internal sensor, or a point sensor. An if expression determines which kind of sensor is treated and calls a corresponding function which performs the actual construction of the STS for the concrete kind of sensor.

8.4 Confidence Conditions

To increase the confidence in the correctness of the generated STSs, confidence conditions regarding the well-formedness of the STSs can be generated and proved. The theory confidence only regards the confidence condition for the route dispatcher:

```
theory confidence :
  axiom
    in
      class
        object N : NETWORK1,  T : TABLES2(N),  G : GENERATOR(T, N)
      end
    ⊢
      S.wf_sts(G.route_dispatcher)
  end
```

Similarly, the confidence conditions can be generated for the other STSs. The theory has not been proved neither formally nor informally, but the STSs have been inspected during specification and found to fulfill the conditions.

8.5 Generator Specification

A model of a generator is specified. The generator produces configuration data for a specific tramway system, based on a description of the system formulated in the DST-L. The model is actually the same model containing the concrete STSs, but by parameterizing this model with the NETWORK and TABLES schemes it implicitly is a generator.

The generator translates the identifiers of the objects in the DST-L to the identifiers of the corresponding objects in the STS-L. Furthermore the generator translates point settings and signals aspects from the description to the corresponding states in the STS-L.

```
value
  sensorid_to_stslsensorid : N.SensorId → S.STSLSensorId,

  pointsetting_to_state : T.PointSetting → S.PointState
  pointsetting_to_state(ps) ≡
    case ps of T.STRAIGHT → S.S_STRAIGHT, T.TURN → S.S_TURN end
```

The format of the configuration data produced by the generator must be structured such that the generic control system easily can access necessary data when initializing the control components and the initial state space. This means that the generic control system does not need to initialize the control components, the physical environment, or the state space, as this initialization is performed by the generator.

8.6 Chapter Summary

A short introduction to the models of generic control components and the physical environment as introduced in chapter 6 has been presented. Furthermore, an initial state space has been specified. It has also been presented how the generator constructs the configuration data for the control system.

Chapter 9

Safety Requirements

In section 2.2 it was stated that a safe tramway network is a network without danger of collisions and derailments. In this chapter the requirements for a safe network are formulated more explicitly and a predicate denoting the requirements is modelled.

The necessary conditions for the tramway network to be safe are given as follows:

- No two trams on the same segment driving towards each other.
- No two trams driving towards the same sensor from opposite directions.
- On each segment the number of trams driving it is less than *max*, a predefined upper limit. This is different from railways as flank protection is not considered necessary. In the predicate *max* has been set to one[1]. Please, note that this is not valid for segments with an entry border sensor.
- No trams on a junction with a controllable point in undefined position.
- No trams at the same time on segments which are crossing.

Thus the safe predicate can be stated as;

```
value
  safe : S.StateSpace → Bool
  safe(ss) ≡
    opposite_direction_on_segment(ss) ∧
    opposite_direction_towards_sensor(ss) ∧
    max_number_on_segment(ss) ∧
    no_trams_on_undefined_points(ss) ∧ trams_in_crossing(ss),

  opposite_direction_on_segment : S.StateSpace → Bool
  opposite_direction_on_segment(ss) ≡
   (
    ∀ s1, s2 : N.SensorId •
      T.path(s1, s2) ⇒
       (
        let
          stsl_s1 = G.sensorid_to_stslsensorid(s1),
          stsl_s2 = G.sensorid_to_stslsensorid(s2),
          (c1, d1) = S.sensoraux(ss)(stsl_s1),
          (c2, d2) = S.sensoraux(ss)(stsl_s2)
        in
```

[1] The control system as it has been modelled only allows one tram at a route at a time, thus *max* is set correspondingly.

```
        d1 = S.sensorid(stsl_s2) ⇒ d2 ≠ S.sensorid(stsl_s1)
      end
     )
 )

opposite_direction_towards_sensor : S.StateSpace → Bool
opposite_direction_towards_sensor(ss) ≡
 (
  ∀ s1, s2 : N.SensorId •
    let
      stsl_s1 = G.sensorid_to_stslsensorid(s1),
      stsl_s2 = G.sensorid_to_stslsensorid(s2),
      (c1, d1) = S.sensoraux(ss)(stsl_s1),
      (c2, d2) = S.sensoraux(ss)(stsl_s2)
    in
      (d1 ≠ S.ε ∧ d1 = d2) ⇒ s1 = s2
    end
 )

max_number_on_segment : S.StateSpace → Bool
max_number_on_segment(ss) ≡
 (
  (all s1, s2 : N.SensorId •
   ~ T.entry_border(s1) ∧
   T.path(s1, s2) ∧
   ~(∃ p : N.PointId • {s1, s2} ∈ {N.stem(p), N.straight(p), N.turn(p)})
    ⇒
     (
      let
        stsl_s1 = G.sensorid_to_stslsensorid(s1),
        stsl_s2 = G.sensorid_to_stslsensorid(s2),
        (c1, d1) = S.sensoraux(ss)(stsl_s1),
        (c2, d2) = S.sensoraux(ss)(stsl_s2)
      in
        abs (c1 − c2) ≤ 1
      end
     )
  )
  ∧
  (all p : N.PointId •
    let
      stsl_stem = G.sensorid_to_stslsensorid(N.stem(p)),
      stsl_straight = G.sensorid_to_stslsensorid(N.straight(p)),
      stsl_turn = G.sensorid_to_stslsensorid(N.turn(p)),
      (c1, d1) = S.sensoraux(ss)(stsl_stem),
      (c2, d2) = S.sensoraux(ss)(stsl_straight),
      (c2, d2) = S.sensoraux(ss)(stsl_straight),
    in
      abs c1 − (c2 + c3) ≤ 1
    end
  )
 )

no_trams_on_undefined_points : S.StateSpace → Bool
no_trams_on_undefined_points(ss) ≡
 (
```

```
  ∀ p : N.PointId •
    let (act, req) = S.pointstates(ss)(G.pointid_to_stslpointid(p)) in
      act ∈ {S.S_MOVE_TO_TURN, S.S_MOVE_TO_STRAIGHT} ⇒
       let
         (c1, d1) = S.sensoraux(ss)(G.sensorid_to_stslsensorid(N.stem(p))),
         (c2, d2) = S.sensoraux(ss)(G.sensorid_to_stslsensorid(N.straight(p))),
         (c3, d3) = S.sensoraux(ss)(G.sensorid_to_stslsensorid(N.turn(p)))
       in
         c1 = c2 + c3
       end
    end
 )

trams_in_crossing : S.StateSpace → Bool
trams_in_crossing(ss) ≡
 (
  ∀ seg1, seg2 : N.Segment •
    N.crossing(seg1, seg2) ⇒
      let
        (s1, s2, s3, s4) : N.SensorId × N.SensorId × N.SensorId × N.SensorId •
        {s1, s2} = N.sensors_of(seg1) ∧ {s3, s4} = N.sensors_of(seg2)
      in
        let
          (c1, d1) = S.sensoraux(ss)(G.sensorid_to_stslsensorid(s1)),
          (c2, d2) = S.sensoraux(ss)(G.sensorid_to_stslsensorid(s2)),
          (c3, d3) = S.sensoraux(ss)(G.sensorid_to_stslsensorid(s3)),
          (c4, d4) = S.sensoraux(ss)(G.sensorid_to_stslsensorid(s4))
        in
          c1 = c2 ∨ c3 = c4
        end
      end
 )
```

Chapter 10

Verification

In this chapter the verification procedure used in this thesis is presented.

The method is presented in [HP00] and applied successfully in [Gja02] and [LV00].

The obligations presented in this chapter have not been formally proved, but an informal argumentation of the correctness is presented in appendix D. Please refer to appendix C for the full specification of the proof obligations.

10.1 Strengthening the Invariant

The goal is to prove that all states[1] reachable from the initial state of the system are safe. The initial state is generated from a description in the DST-L and must first be proved to be safe.

Proving that all reachable states are safe amounts to proving that every possible application of the interpret_ rule function (the only function modifying the global state) preserves safety. The possible applications correspond to the conditions and actions of the generated STSs modelling the control components and physical environment.

Furthermore, because the tramway system, being the control components and physical devices, is based on the parallel execution of STSs it must be verified that the order of firing rules from the STSs does not influence the system.

Thus, the following must be fulfilled for the tramway system to be safe:

1. The description in the DST-L must conform to the actual physical network.
2. The initial state generated from the description is safe, assuming the description is correct.
3. All states reachable from the initial state are safe.
4. The parallel execution of STSs is safe.

This proof is not possible to construct inductively, since the interpret_rule function may be applied correctly to a safe state not reachable from the initial state, and yield a state that is not safe. For a simple example of a safe state that could become unsafe see figure 10.1. If one of the trams drove past the sensor, the state would become unsafe. Hence, the safe predicate cannot be shown to be an invariant of the system as illustrated in figure 10.2.

[1] In this chapter **state** denotes the global state of the system, called *StateSpace* in the model.

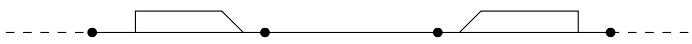

Figure 10.1: Safe, but not for long.

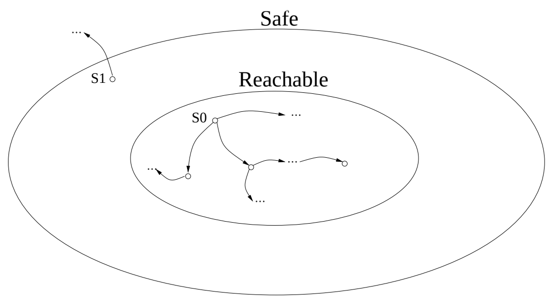

Figure 10.2: Safe cannot be proved to be an invariant.

Therefore, the *invariant is strengthened* by introducing a predicate consistent. The idea is to be able to prove inductively that the interpret_rule function applied to a consistent state will always yield a consistent state as illustrated in figure 10.3. Thus, consistent is an invariant of the system when started in the initial state. By requiring that consistent implies safe, safe is indirectly proved to be an invariant of the system when started in the initial state. If it was not required that a consistent state is also safe, it would be possible to prove consistent to be an invariant of the system, but it would not be possible this way to prove safe to be an invariant of the system, as illustrated in figure 10.4.

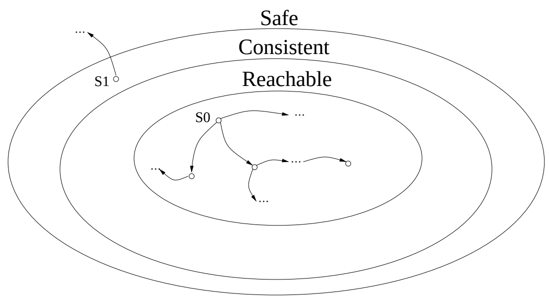

Figure 10.3: Consistent is an invariant and implies safe.

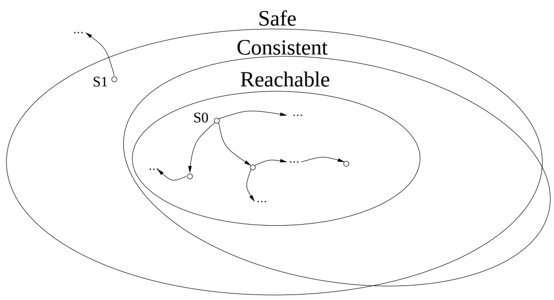

Figure 10.4: Consistent is an invariant, but does not imply safe.

10.2 The *consistent* Predicate

The consistent predicate is introduced. This predicate is more restrictive than the safe predicate, and introduces constraints on the values of devices, when the routes should be reserved, and so forth. The consistent predicate can be found in appendix C.1, and parts of it will be commented on in the following. The predicate is a conjunction of several predicates.

```
value
  consistent : S.StateSpace → Bool
  consistent(ss)≡
    safe(ss) ∧
    consistent_reservations(ss) ∧
    consistent_control_counter_values(ss) ∧
    consistent_counter_values(ss) ∧
    consistent_counter_flow_route(ss) ∧
    consistent_control_counter_flow_route(ss) ∧
    legal_signal_values(ss) ∧ legal_point_values(ss)
```

10.2.1 *safe*

The *safe* predicate is described in chapter 9.

10.2.2 *consistent_reservations*

The reservations must be consistent, meaning that two conflicting routes must not be reserved at the same time.

```
value
  consistent_reservations : S.StateSpace → Bool
  consistent_reservations(ss)≡
   (
    ∀ r1, r2 : T.RouteId •
      r1 ≠ r2 ∧
      let
        (req1, res1) = S.rcvstates(ss)(G.routeid_to_rcvid(r1)),
        (req2, res2) = S.rcvstates(ss)(G.routeid_to_rcvid(r2))
      in
        res1 ∧ res2 ⇒∼ T.conflicting(r1, r2)
      end
   )
```

10.2.3 *consistent_control_counter_values*

Sensors that are *high* or with a value of the control counter larger than zero, must be part of a reserved route. The exception is entry border sensors which are not restricted in this way.

```
value
  consistent_control_counter_values : S.StateSpace → Bool
  consistent_control_counter_values(ss)≡
   (
```

```
    ∀ s : N.SensorId •
      let cc = S.ccountstates(ss)(G.sensorid_to_ccountid(s)) in
        ~ T.entry_border(s) ∧
        (cc > 0 ∨ S.sensorstates(ss)(G.sensorid_to_stslsensorid(s)) = S.S_HIGH) ⇒
         (
          ∃ r : T.RouteId •
           let (req, res) = S.rcvstates(ss)(G.routeid_to_rcvid(r)) in
             res ∧ s ∈ elems T.route_sensors_of(r)
           end
         )
      end
  )
```

10.2.4 *consistent_counter_values*

The values of sensor counters must be consistent. This predicate is a conjunction of three predicates each defining consistency for a counter of a sensor with different properties. Basically, if the flow out of a sensor in one direction is larger than zero, the sensor must be part of a reserved route. Please note that border sensors do not have predicates on consistent counter values.

```
value
  consistent_counter_values : S.StateSpace → Bool
  consistent_counter_values(ss) ≡
    consistent_counter_values_internal(ss) ∧
    consistent_counter_values_stem(ss) ∧
    consistent_counter_values_turn_straight(ss)
```

consistent_counter_values_internal

Consistency for the counter of an internal sensor means that if the counter of the considered sensor is larger than one of the counters of its neighbouring sensors it must be part of a reserved route. Entry border sensors are explicitly excluded, while exit border sensors are excluded as their predicate for consistent route flow determines that they cannot have values of counters larger than their neighbour.

```
value
  consistent_counter_values_internal : S.StateSpace → Bool
  consistent_counter_values_internal(ss) ≡
   (
    ∀ s1, s2 : N.SensorId •
      let
        (c1, d1) = S.sensoraux(ss)(G.sensorid_to_stslsensorid(s1)),
        (c2, d2) = S.sensoraux(ss)(G.sensorid_to_stslsensorid(s2))
      in
        T.path(s1, s2) ∧
        ~ T.entry_border(s1) ∧
        c1 > c2 ∧ N.point_at(s1) = N.none ∧ N.point_at(s2) = N.none ⇒
         (
          ∃ r : T.RouteId •
           let (req, res) = S.rcvstates(ss)(G.routeid_to_rcvid(r)) in
             res ∧ s1 ∈ elems T.route_sensors_of(r)
```

```
        end
      )
    end
  )
```

consistent_ counter_ values_ stem and *consistent_ counter_ values_ turn_ straight* are similar just with a slightly more complicated calculation of the relationship with neighbours.

10.2.5 *consistent_ counter_ flow_ route*

For all routes the counter flow in the sensor sequence defining the route must be consistent. The predicate uses an auxiliary function for recursively checking that the flow out of a sensor in the route direction is larger than or equal to zero for all sensors in the sensor sequence defining the route.

```
value
  consistent_counter_flow_route : S.StateSpace → Bool
  consistent_counter_flow_route(ss) ≡
    (
      ∀ r : T.RouteId •
        consistent_counter_flow_sensorlist(T.route_sensors_of(r), ss)
    )
```

The function that is used for recursively defining the consistent flow in a route uses the knowledge that a route can only pass a certain way through a point, meaning that the stem sensor of a controllable point in a route is always followed by the turn or straight sensor.

```
value
  consistent_counter_flow_sensorlist : N.SensorId* × S.StateSpace → Bool
  consistent_counter_flow_sensorlist(sl, ss) ≡
    let (c0, d0) = S.sensoraux(ss)(G.sensorid_to_stslsensorid(hd sl)) in
      if ∃ p : N.PointId • hd sl = N.stem(p) ∧ N.controllable(p) then
        let
          p : N.PointId • hd sl = N.stem(p) ∧ N.controllable(p),
          (c2, d2) = S.sensoraux(ss)(G.sensorid_to_stslsensorid(N.straight(p))),
          (c3, d3) = S.sensoraux(ss)(G.sensorid_to_stslsensorid(N.turn(p)))
        in
          c0 ≥ c2 + c3
        end
      elsif
        ∃ p : N.PointId • hd sl ∈ {N.straight(p), N.turn(p)} ∧ ~ N.controllable(p)
       then
         let
           p : N.PointId • hd sl ∈ {N.straight(p), N.turn(p)} ∧ ~ N.controllable(p),
           (c1, d1) = S.sensoraux(ss)(G.sensorid_to_stslsensorid(N.stem(p))),
           (c2, d2) = S.sensoraux(ss)(G.sensorid_to_stslsensorid(N.straight(p))),
           (c3, d3) = S.sensoraux(ss)(G.sensorid_to_stslsensorid(N.turn(p)))
         in
           c2 + c3 ≥ c1
         end
      else
        let (c4, d4) = S.sensoraux(ss)(G.sensorid_to_stslsensorid(hd tl sl)) in
          c0 ≥ c4
        end
      end
```

```
  end ∧
  (len tl sl > 1 ⇒ consistent_counter_flow_sensorlist(tl sl, ss)),
```

10.2.6 *consistent_control_counter_flow_route*

The control counter flow in routes must also be consistent. The predicate is almost identical to *consistent_counter_flow_route* except that it uses control counters instead of sensor counters for calculating the flow.

10.2.7 *legal_signal_values*

The values of signals must be consistent, meaning that signals can only show *go* or be requested to show *go* if a route starting at the sensor where the signal is located is reserved.

```
value
  legal_signal_values : S.StateSpace → Bool
  legal_signal_values(ss) ≡
   (
    ∀ sig : N.SignalId •
      let (act, req) = S.signalstates(ss)(G.signalid_to_stslsignalid(sig)) in
        act = S.S_GO ∨ req = S.S_GO ⇒
         (
          ∃ r : T.RouteId •
           let (rreq, rres) = S.rcvstates(ss)(G.routeid_to_rcvid(r)) in
             rres ∧ N.sensor_of(T.route_start(r)) = sig
           end
         )
      end
   )
```

10.2.8 *legal_point_values*

If the route is reserved and the entry sensor of the route has been passed, all points on the route must be set to and requested to be the value given by the pointsetting table.

```
value
  legal_point_values : S.StateSpace → Bool
  legal_point_values(ss) ≡
   (
    ∀ r : T.RouteId •
     let
       (req, res) = S.rcvstates(ss)(G.routeid_to_rcvid(r)),
       start_controlcounter =
        S.ccountstates(ss)(G.sensorid_to_ccountid(T.route_start(r)))
     in
       res ∧ start_controlcounter > 0 ⇒
        (
         ∀ p : N.PointId • p ∈ T.points_at(r) ⇒
          let (actp, reqp) = S.pointstates(ss)(G.pointid_to_stslpointid(p)) in
            actp = G.pointsetting_to_state(T.pointsetting_of_point_in_route(p, r)) ∧
            actp = reqp
          end
```

```
        )
      end
    )
```

10.3 Strong Safety Axioms

Proving the consistent predicate to be an invariant of the system when started in initial state amounts to proving a number of axioms, called the strong safety axioms, following the inductive principles presented in section 10.1. In this section the different axioms are presented.

10.3.1 [initial_statespace_consistent]

The initial state space generation from a description in the DST-L must be consistent:

```
theory zeroth_strong :
  axiom
    [initial_statespace_consistent]
      in
        class
          object N : NETWORK1, T : TABLES2(N), G : GENERATOR(T, N)
        end
      ⊢
        (in CONSISTENT(G, T, N) ⊢ consistent(G.initial_statespace))
  end
```

10.3.2 [consistent_is_safe]

To be able to deduct from the fact that consistent is an invariant of the system, that safe is also an invariant of the system, a state space that is consistent must also be safe.

```
theory first_strong :
  axiom
    [consistent_is_safe]
      in
        class
          object N : NETWORK1, T : TABLES2(N), G : GENERATOR(T, N)
        end
      ⊢
        (in CONSISTENT(G, T, N) ⊢ ∀ φ : S.StateSpace • consistent(φ) ⇒ safe(φ))
  end
```

10.3.3 [interpret_STS_list_consistent]

Corresponding to the inductive step of proving that consistent is an invariant, the interpret_rule function applied to a consistent state must yield a consistent state.

There is an axiom for each STS type, which for physical devices contains predicates modelling the timing assumptions of the system. Thus, for instance may_interpret_sensor excludes certain states which are consistent, but are excluded by timing assumptions.

The description of the timing assumptions is in section 10.4, the axioms are in appendix C.2, and the predicates modelling timing assumption are in C.1.

A generic second strong axiom for an STS control component or an STS of the physical environment STSs can be given as:

```
theory second_strong :
  axiom
    [interpret_STSs_consistent]
      in
        class
          object N : NETWORK1, T : TABLES2(N), G : GENERATOR(T, N)
        end
      ⊢
        (
          in CONSISTENT(G, T, N) ⊢
            ∀ φ : S.StateSpace, i : Nat •
              i > 0 ∧
              i ≤ len S.STSids ∧
              consistent(φ) ∧ may_interpret_STS(φ, G.STSs(i)) ⇒
              consistent(S.interpret_sts(G.STSs(i))(φ))
        )
```

All instances of STS in the generic axiom above can be replaced with a component to yield a specific axiom, for instance counter_controllers. The exception is the S.STSids where STS must be replaced with the id defining the number of the specific type of component.

Please note that since only one route dispatcher exists for a given control system the axiom is slightly simpler, as it does not concern a list of STSs. Furthermore, timing assumptions only exist for physical devices meaning that for control components the may_interpret_STS part is not in the axiom.

10.3.4 [parallel_STS1_STS2_consistent]

Several physical devices are used in the system, being sensors, signals, and points. Furthermore, a computer runs the control system. The three distinct types of physical entities are actual physical devices, making the parallel combination of the physical entities a model of true concurrency. The control components of the control system on the other hand are software entities running on the same computer and are interleaved in the control algorithm (interpreter).

Therefore, for the system to be safe it is necessary to prove that two STSs simulating concurrency, are not affected by the order of execution. Thus, if the first STS fires a transition rule, the second should still be able to fire transition rules with conditions that were fulfilled before the first transition rule was fired.

Imagine the scenario where two STS running in parallel each has a rule that fires when interpreted, with the following interleaving of processing data:

- The first STS interprets its rule, thereby evaluating the conditions to ”true”.
- The second STS interprets its rule, evaluating the conditions to ”true”, and processes the actions, thereby updating the state. Unfortunately, the conditions of the rule of the first STS no longer evaluate to ”true” in the post state.
- The first STS now processes the actions as the conditions were evaluated to ”true”, even though this is no longer the case.

As the control algorithm interleaves the execution of the control components it is not necessary to prove that a parallel execution of two control components preserves consistency. Therefore, for each combination of control component and STSs of physical environment, and two STSs of the physical environment an axiom is introduced.

The axioms are in appendix C.2.

A generic third strong axiom for the STSs STS1 and STS2 is[2]:

```
theory third_strong :
  axiom
    [parallel_STS1s_STS2s_consistent]
      in
        class
          object N : NETWORK1, T : TABLES2(N), G : GENERATOR(T, N)
        end
      ⊢
        (
          in CONSISTENT(G, T, N) ⊢
            ∀ φ : S.StateSpace, (s1, s2) : S.Sts × S.Sts, (r1, r2) : S.Rule × S.Rule •
              let
                (ss1, result1) = S.interpret_rule(r1)(φ),
                (ss2, result2) = S.interpret_rule(r2)(φ)
              in
                consistent(φ) ∧
                s1 ∈ elems G.STS1s ∧
                s2 ∈ elems G.STS2s ∧
                r1 ∈ elems s1 ∧
                r2 ∈ elems s2 ∧
                may_interpret_STS1(φ, s1) ∧
                may_interpret_STS2(φ, s2) ∧ result1 ∧ result2 ⇒
                let
                  (ss3, result3) = S.interpret_rule(r1)(S.interpret_sts(s1)(φ)),
                  (ss4, result4) = S.interpret_rule(r2)(S.interpret_sts(s2)(φ))
                in
                  result3 ∧ result4
                end
              end
        )
```

Please note that timing assumptions only exist for physical devices meaning that for control components the may_interpret_STS part is not present in the axiom.

The specific interaction between the different STSs can be seen in figure 10.5. The columns are the different STS: *RD* denotes the route dispatcher, *RCs* route controllers, *CCs* counter controllers, *Sigs* signals, *Ps* points, and *Sens* sensors. The rows are the different kinds of variables. A notation guide was introduced in section 6.2 in figure 6.1.

R means the STS reads that kind of variable, and *W* means that it is written by the STS. It is only STSs that read and/or write common variables that can cause problems with the parallel execution, and as can be seen from the figure the parallel execution of several combinations of two STSs are trivially safe as they do not read/write common variables.

[2]Please note that the axioms concerning the route dispatcher is slightly simpler.

	Control Components			Physical Environment		
	RD	*RCs*	*CCs*	*Sigs*	*Ps*	*Sens*
rcv	R W	R W				
a(S)	R	R		R W		
a(W)		R			R W	
r(S)		W		R		
r(W)		W			R	
v(G)			R			R W
c(G)	R					R W
cc(G)		R W	W			
d(G)						R W
s(G)						R

Figure 10.5: Read and Write relation between STSs and variables.

10.4 Timing Assumptions

To be able to prove the proof obligations certain assumptions considering the timing of physical events are made. As stated in section 10.3.4 the control components are interpreted in order while the STSs of the physical environment are a case of true concurrency.

The reason that these assumptions are necessary is that the model allows reachable states that are not consistent as illustrated in figure 10.6. In certain consistent reachable states it is possible to fire a transition causing a state change to one of the inconsistent states (denoted *"Unwanted Event"* in the figure). The assumptions postulate that such an *"Unwanted Event"* will not occur as the system will stay in one of the states denoted *Reachable Physical*. In the proof obligations predicates have been added for sensors expressing when a transition is physical possible, thereby, excluding as some transitions as marked in the figure.

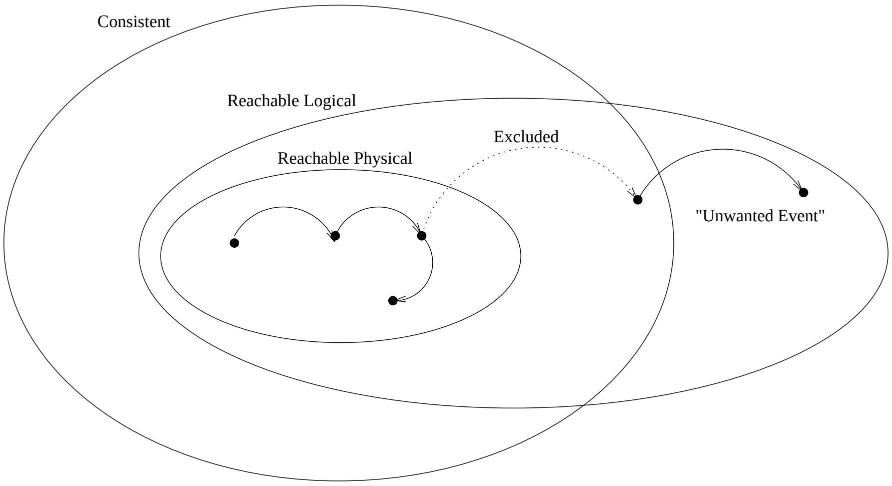

Figure 10.6: Assumptions and consistency.

The assumptions are presented in the following and the predicates (see appendix C.1) are modelled stating when the STSs simulating the devices can be interpreted corresponding to the assumptions. In the end of the section it is argued that the timing assumptions are reasonable.

Certain definitions are used in the assumptions:

- A cycle of running the control algorithm of the control system t_{cycle}.
- The cycles of running the different devices are t_{sensor_cycle}, t_{signal_cycle}, and t_{point_cycle} respectively[3].
- Tram minimum length l_{min}.
- Maximum speed of trams v_{max}.
- The minimum time a sensor must stay high $t_{sensor\_min\_high}$.
- The maximum time a sensor must stay high $t_{sensor\_max\_high}$.

1: A sensor must become low before next tram arrives at the sensor. If this is not the case two or more trams may pass a sensor while it is *high* and the control system will think that only one tram has passed.

For the next tram to pass the sensor the first tram must have fully passed the sensor. *Thus, the sensor must only be high for at most as long as it takes a tram to pass it* :

$$t_{sensor\_max\_high} < \frac{l_{min}}{v_{max}} \tag{10.1}$$

This basic assumption need not be modelled in the may_interpret_predicates.

2: A sensor cannot become low before the control counter has been increased. The STSs simulating the sensors are not restricted in going from *high* to *low*. Therefore, the counter controller has to increase the control counter before the sensor becomes *low*. Otherwise, a tram may enter a route without the route controller being able to detect it, which is definitely not desirable.

As one control component must be interpreted it is assumed that it takes one cycle of the control algorithm to update the control counter. *Thus, for the assumption to be reasonable the sensor must stay high for at least one cycle of the control algorithm.* :

$$t_{sensor\_min\_high} > t_{cycle} \tag{10.2}$$

This implies that transition rule B of figure 10.8 is fired before transition rule C of figure 10.7.

This is modelled in the predicate sensor_may_become_low which is part of the predicate may_interpret_sensors modelling when STSs simulating sensors may be interpreted:

```
sensor_may_become_low : S.StateSpace × S.Sts ~→ Bool
sensor_may_become_low(ss, sts)≡
  let
    i : Nat • sts = G.sensor_simul(i),
    sid = N.sensorids(i),
    senval = S.sensorstates(ss)(G.sensorid_to_stslsensorid(sid))
  in
    senval = S.S_HIGH⇒
    let cc_cmv = G.ccountid_to_cmvid(G.sensorid_to_ccountid(sid)) in
      S.cmvstates(ss)(cc_cmv) = S.COUNTED
    end
  end
```

[3]The cycle time of devices is the time it takes to interpret the STS.

The predicate basically states that if the sensor is *high*, the STS simulating the sensor may be interpreted if the control mode variable of the control counter indicates that it has been increased.

3: The control counter must be ready to count before the tram has passed the sensor. If the sensor becomes *high* before the counter controller is ready to count again, the control system may not have an accurate count of how many trams are on the segment.

When the front of the tram passes the sensor, the sensor becomes high. Thereafter, the control counter is increased, and the sensor becomes low. At this point it takes one cycle of the counter controller to become ready to count. *Thus, for the assumption to be reasonable first a cycle of the control system is needed, then a cycle of the sensor simulation, and finally a cycle of the control system before the tram has completely passed the sensor:*

$$2 \cdot t_{cycle} + t_{sensor\_cycle} < \frac{l_{min}}{v_{max}} \tag{10.3}$$

This implies that transition rule D of figure 10.8 is fired before transition rule A of figure 10.7.

This is modelled in the predicate sensor_low_control_counter_ready which is part of the may_interpret_sensor predicate:

```
sensor_low_control_counter_ready : S.StateSpace × S.Sts ~→ Bool
sensor_low_control_counter_ready(ss, sts) ≡
  let
    i : Nat • sts = G.sensor_simul(i),
    sid = N.sensorids(i),
    senval = S.sensorstates(ss)(G.sensorid_to_stslsensorid(sid))
  in
    senval = S.S_LOW⇒
    let cc_cmv = G.ccountid_to_cmvid(G.sensorid_to_ccountid(sid)) in
      S.cmvstates(ss)(cc_cmv) = S.READY_TO_COUNT
    end
  end
```

The predicate basically states that if the sensor is *low*, the STS simulating the sensor may be interpreted if the control mode variable of the control counter indicates that it is ready to count.

4: If a signal is present it must show stop before the tram has passed the sensor. A route controller requests that the entry signal of the route is set to *stop*, once a tram starts the passing of the start sensor of the route. To prevent additional trams entering the route, the signal must show *stop* before the tram has completely passed the sensor.

When the tram passes the start sensor of the route, the counter controller must notice that the sensor is *high* and increase the control counter. Then the route controller must notice that the control counter is larger than 1 and request the shifting of the signal. Finally, the signal must become *stop*. *Thus, two cycles of the control system and one cycle of the signal is required before the tram has passed the sensor:*

$$2 \cdot t_{cycle} + t_{signal\_cycle} < \frac{l_{min}}{v_{max}}$$

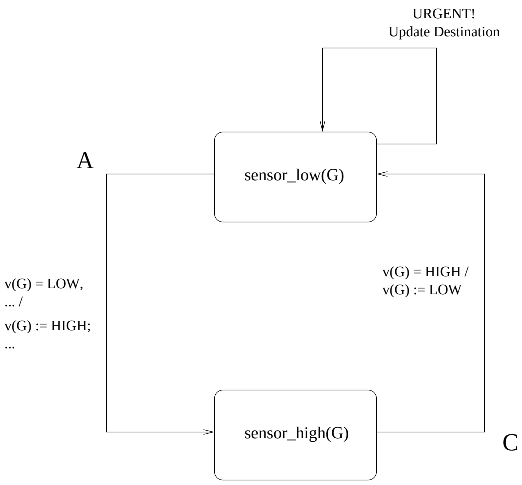

Figure 10.7: Simplified STS of sensor G.

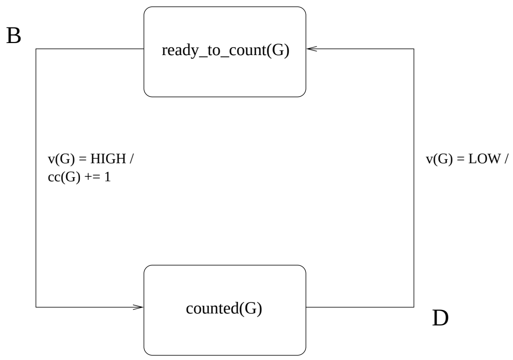

Figure 10.8: Counter controller for sensor G.

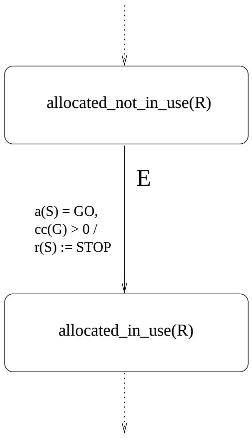

Figure 10.9: Route controller for route R.

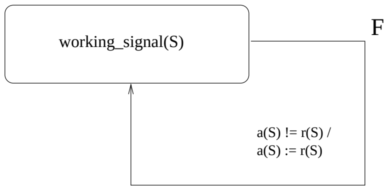

Figure 10.10: STS of signal S.

This implies the following sequence of firing transition rules: B of figure 10.8, E of figure 10.9, and F of figure 10.10.

This is modelled in the predicate sensor_high_signal_updated which is part of the may_interpret_sensor predicate. The predicate is more restrictive than the assumption, as it is not possible to express the state space in which the STS simulating the sensor may be interpreted. This is because such a predicate would demand that when the sensor is *low* it may only be interpreted if the signal shows *stop*. This a direct contradiction to the conditions of STSs simulating sensors and therefore it is not usable. Therefore, the predicate indicates that the signal must show stop before the sensor becomes low.

```
sensor_high_signal_updated : S.StateSpace × S.Sts ~→ Bool
sensor_high_signal_updated(ss, sts) ≡
  let
    i : Nat • sts = G.sensor_simul(i),
    sid = N.sensorids(i),
    senval = S.sensorstates(ss)(G.sensorid_to_stslsensorid(sid))
  in
    senval = S.S_HIGH ∧ N.signal_at(sid) ≠ N.none ⇒
    let
```

```
    sig : N.SignalId • N.sensor_of(sig) = sid,
    (reqsig, actsig) = S.signalstates(ss)(G.sensorid_to_stslsensorid(sig))
  in
    actsig = S.S_STOP
  end
end
```

This corresponds to that the minimum time the sensor must stay high is larger than two cycles of the control system and one cycle of the signal:

$$t_{sensor\_min\_high} > 2 \cdot t_{cycle} + t_{signal\_cycle} \tag{10.4}$$

5: The destination of sensors must be updated to ϵ as soon as no trams are headed towards the destination sensor. If this is not the case, the safe predicate may return false, even though, the network is actually safe. This is due to the predicate opposite_direction_towards_sensor stating that two sensors cannot have the same destination. If the destination is not reset, another tram may arrive at a later time, and this part of the safety predicate may become false. Consequently, the destination of a sensor must be ϵ before the tram has completely passed the sensor of the destination, corresponding to that the transition rule in figure 10.7 marked urgent must occur before the passing is completed.

As soon as the destination sensor becomes high, the sensor can fire the transition rule resetting the destination. *Thus, one cycle of the sensor must pass before the tram has completely passed the sensor*:

$$t_{sensor\_cycle} < \frac{l_{min}}{v_{max}}$$

Similarly to the predicate sensor_high_signal_updated, the predicate destination_ok is more restrictive than the desired assumption:

```
destination_ok : S.StateSpace × S.Sts ~→ Bool
destination_ok(ss, sts) ≡
  let
    i : Nat • sts = G.sensor_simul(i),
    sid = N.sensorids(i),
    senval = S.sensorstates(ss)(G.sensorid_to_stslsensorid(sid))
  in
    senval = S.S_HIGH ⇒
     let
       neighbours = { senid | senid : N.SensorId • T.path(senid, sid)}
     in
       (all senid2 : N.SensorId • senid2 ∈ neighbours =
         let
          (c, d) = S.sensoraux(ss)(G.sensorid_to_stslsensorid(senid2))
         in
           d ≠ S.sensorid(G.sensorid_to_stslsensorid(sid))
         end
       )
     end
  end
```

The predicate indicates that the destination must be reset before the sensor becomes *low*. *Thus, in this case the sensor must stay high at least one sensor cycle:*

$$t_{sensor\_min\_high} > t_{sensor\_cycle} \tag{10.5}$$

Please note that this list of assumptions is not necessarily exclusive, as the proofs have not been conducted formally, hence certain necessary assumptions may be missing.

That the assumptions are reasonable can be deducted from the following data from the domain:

- A cycle of running the control algorithm of the control system is approximately 100 ms with a representative network size[4].
- A tram has a minimum length of 20 m, as presented in [Eng03].
- The maximum speed of the trams is 70 km/h, as presented in [Eng03].

If it is assumed that the cycle time of each of the devices is equal to the cycle time of the control system[5], the following time bounds corresponds to the data above:

$$\begin{array}{lcrl}
(10.1) & t_{sensor_max_high} & < & 0.97 \text{ s} \\
(10.2) & t_{sensor_min_high} & > & 0.1 \text{ s} \\
(10.3) & 0.3 \text{ s} & < & 0.97 \text{ s} \\
(10.4) & t_{sensor_min_high} & > & 0.3 \text{ s} \\
(10.5) & t_{sensor_min_high} & > & 0.1 \text{ s}
\end{array}$$

Some of the assumptions pose demands on the sensor hardware. Identifying such sensors is beyond the scope of this thesis, but assuming they are readily available the time calculations above clearly indicate that the assumptions are reasonable.

Obviously, t_{cycle} is dependent on the size of the tramway network. Thus, for larger than normal network sizes it is necessary to generate more than one control system.

10.5 Chapter Summary

In this chapter the verification procedure of the thesis has been introduced, with predicates used to strengthen the invariant. Furthermore, assumptions about the timing of physical events have been introduced and argumentation has been presented stating that they are valid.

[4]This data is obtained from Jan Peleska.

[5]This is an upper bound for the cycle time of the devices, as they are considered to be faster than the control system

Chapter 11

Development of Imperative Models

In this chapter a transformation of the applicative models to imperative models is presented. This is done to facilitate the implementation of the models. The models affected by the development are the operational semantics of the STS-L presented in section 7.5 and the generator model presented in chapter 8.

11.1 Imperative STS-L Operational Semantics

The transformation of the STS-L operational semantics to an imperative version results in a transformation of the types of interest, the observers, and the generators. The resulting model is in the RSL scheme I_STS_Operational_Semantics which can be found in appendix E.1.

11.1.1 Transformation of Types of Interest

The only type of interest in the STS-L model is the state space. Object arrays are introduced for each map of the state space, and local varibles are introduced for each part of their range. Functions are added to each object corresponding to the retrieval and overriding functionality of maps. If the range of the map is a cartesian product, two variables exist in the object, each with their own retrieval and overriding functionality.

The functions of the applicative model corresponding to the different possible manipulation of variables have been specified using the above mentioned functions instead of the map functionality. Thus, the set_... and get_... of the operational semantics functions are specified using the functions of the object arrays.

```
object
  SEN[sen : STSLSensorId] :
    hide sensor_state_var in
      class
        variable sensor_state_var : SensorState

      value
        get_sensor_state_var : Unit → read any SensorState
        get_sensor_state_var()≡sensor_state_var,
```

```
    set_sensor_state_var : SensorState → write any Unit
    set_sensor_state_var(ss) ≡ sensor_state_var := ss
  end
```

The variables are not initialized in the objects, thereby, allowing more than one initial state space.

Modelling the state space as object arrays is done to facilitate an object oriented implementation, and provides a structure that can be represented reasonably in an imperative implementation language.

11.1.2 Transformation of Observers and Generators

Observers in the STS-L operational semantics have the general form:

```
value
  observer : STSLId → StateSpace → STSLId
```

According to [GHH+95], when transformed to imperative style, this becomes:

```
value
  observer : STSLId → read any STSLId
```

Thus, for instance the imperative version of *eval_cond* can be specified as:

```
value
  eval_cond : Condition ∼→ read any Bool
  eval_cond(cond) ≡
    let s1 = eval_idcomp(idcomp_of(cond)), s2 = eval_expr(expr(cond)) in
      case relation(cond) of
        REL_LE → count(s1) ≤ count(s2),
        REL_GT → count(s1) > count(s2),
        REL_LT → count(s1) < count(s2),
        REL_EQUAL → s1 = s2,
        REL_NOT_EQUAL → s1 ≠ s2
      end
    end
  pre wf_cond(cond),
```

Similarly, generator functions has the general form;

```
value
  generator : STSLId × State → StateSpace → StateSpace
```

This becomes[1]:

```
value
  generator : STSLId × State → read any write any Unit
```

Thus, the imperative version of *process_action* can be specified as;

```
value
  process_action : Action ∼→ read any write any Unit
  process_action(act) ≡
    let new_state = eval_expr(expr(act)) in
```

[1]Can be shortened to **write any Unit**, but this is not done to make the functions more readable

```
case operation(act) of
  OP_PLUS_SET→
    case idcomp_of(act) of
      mk_counterSen(sd) → plus_set_sensor_counter_state(sd, count(new_state)),
      ...
    end,
  OP_SET→
    case idcomp_of(act) of
      mk_actualSen(sd) → set_actual_sensor_state(sd, sensorstate(new_state)),
      ...
          end
end
  end
pre wf_act(act),
```

This method for transformation is generic and is used on all observers and generators.

Development Relation

The relation is a *static change*, as the signatures of values have changed as defined in [GHH+95]. Nevertheless, the development has been systematic as explained above.

11.2 Refinement of Imperative STS-L Operational Semantics

The imperative version of the STS-L operational semantics has been refined in the scheme I2_STS_operational_semantics found in appendix E.2. The development is transforming recursive definitions of functions on lists to iterative definitions.

For instance, the new version of the function interpret_sts :

```
value
  interpret_sts : Sts ~→ read any write any Unit
  interpret_sts(sts)≡
    local variable i : Int := 1 in
      while i ≤ len sts do
        if interpret_rule(sts(i)) then skip else i := i + 1 end
      end ;
      skip
    end
   pre wf_sts(sts),
```

This development has been performed to make the implementation based on the final model as simple and systematic as possible.

Development Relation

The development relation can be expressed:

```
devt_relation STS_operational_semanticsI_I2(
  I2_STS_operational_semantics for I_STS_operational_semantics) :
   ⊢ I2_STS_operational_semantics ≼ I_STS_operational_semantics
```

The bodies of the functions interpret_sts, eval_cond_list, and process_action_list have been changed. Thus, proving this relation amounts to proving that the new definition of the bodies of these functions implies the old definition of the bodies.

Informal arguments are presented for the function interpret_sts stating that it is an implementation of the version in I_STS_operational_semantics. The argumentation for eval_cond_list and process_action_list is similar, but is not presented.

The version of interpret_sts presented in I_STS_operational_semantics (see appendix E.1) can clearly be written as the following as stated in [GHH+95] page 197.

```
value
  interpret_sts : Sts ~→ read any write any Unit
  interpret_sts(sts)≡
    while ( sts ≠ ⟨ ⟩ ) do
      if interpret_rule(hd sts) then skip else interpret_sts(tl sts) end
    end;
    skip
  pre wf_sts(sts)
```

The first imperative version of the interpret_sts interprets one rule and then calls itself on the tail of the sts, and consequently the rules of the sts are interpreted in order, stopping whenever one evaluates to "true". The second imperative version does exactly the same except that it stays in the while loop, thereby interpreting the rules in order, and terminating when one evaluates to true. Thus, it is intuitively clear that both versions interpret the rules in the same order and have the same termination criteria, and it is postulated to be an implementation.

11.3 Imperative Generator

An imperative version of the GENERATOR RSL scheme is modelled. The transformation from applicative is done according to the imperative version of the STS operational semantics. The imperative version is in the RSL scheme I_GENERATOR which can be found in appendix E.4.

Initial state space As the imperative state space is modelled as a number of object arrays, these object arrays are initialized. The initial values of the variables in the object are identical to the initial values used in the applicative model.

STS It would make sense to transform the STS from values to variables in the imperative model. But since the STSs are static in the sense that once created they are not modified, they remain values in the imperative version.

11.3.1 Development Relation

This is not an implementation, as I_GENERATOR uses functionality from the global object I_S and GENERATOR from S. Furthermore, I_GENERATOR initializes the state space according to the form presented in I_STS_operational_semantics[2] which cannot be shown

[2]Actually it is I2_sts_operational_semantics, but the state space has not been changed in this development.

to be an implementation of the state space of STS_operational_semantics. Nevertheless, the development has been systematic, and the imperative model of the generator constitutes generation in the imperative model.

11.4 Chapter Summary

In this chapter the development of the operational semantics of the STS-L from applicative to imperative model has been presented, as well as the development of the generator from applicative to imperative model.

Chapter 12

Specification of Control Algorithm

The control algorithm deciding the order of execution of the different STS control components and of the physical environment presented in section 6.4.4 is specified in this chapter. The control algorithm is in the RSL scheme INTERPRETER which can be found in appendix F.

The control algorithm executes the STSs representing the control components in order as described in 6.4.4. Thus, the interpreter function (control algorithm) executes these STSs in a nonterminating while-loop. The function interpret_sts is from the imperative version of the STS-L and runs through the STS's rules in order, trying to fire one. If a transition rule is fired the function terminates. Otherwise, it terminates when all rules have been interpreted. One run of the while loop is called a cycle of the control system.

```
value
  interpreter : Unit → read any write any Unit
  interpreter() ≡
    G.initial_statespace() ;
    while true do
      for i in ⟨ 1 .. len G.counter_controllers ⟩ do
        I_S.interpret_sts(G.counter_controllers(i))
      end ;
      I_S.interpret_sts(G.route_dispatcher) ;
      for i in ⟨ 1 .. len G.route_controllers ⟩ do
        I_S.interpret_sts(G.route_controllers(i))
      end
    end
```

Chapter 13

Implementing the DST-L

In this chapter the implementation of the Domain Specific Tramway Language (DST-L) is presented.

The implementation is based on the RSL models NETWORK1 (see appendix B.1.2) and TABLES2 (see appendix B.2.3) presented in chapter 5.

It has been chosen to implement the DST-L in the extended markup language (XML) [W3Cb], since XML is a good standard for storing structured data. Furthermore, tools are available for manipulating the data stored in XML documents, and for collecting and presenting data.

This chapter concerns how the RSL specification has been translated to XML.

In section 13.1 the implementation of the abstract syntax and in section 13.2 the implementation of the static semantics is presented. In section 13.3 an application for collection of data is described. Finally, section 13.4 introduces a graphical representation of descriptions in DST-L.

The overall structure of the files containing the implementation of the DST-L is illustrated in figure 13.1. The arrows represent files including other files, the target file includes the source file. The source file of an arrow marked «input» is the input for the target file, and the target file of an arrow marked «output» is the output for the source file.

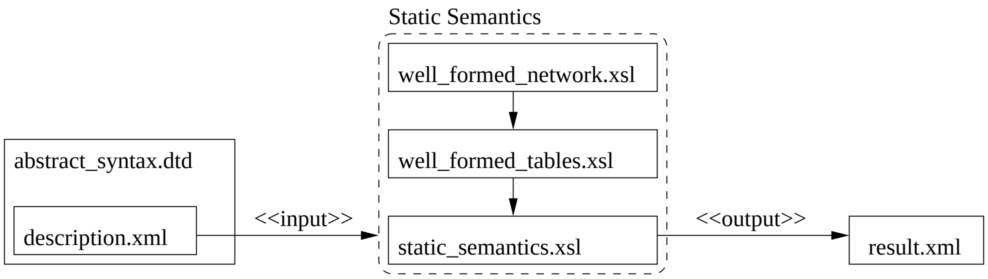

Figure 13.1: Structure of XML implementation of the DST-L.

The file description.xml contains the data of the network description and the interlocking tables. The file abstract_syntax.dtd implements the abstract syntax as described in section 13.1 and the three files well_formed_network.xsl, well_formed_tables.xsl, and static_semantics.xsl implements the static semantics of the DST-L as described in section 13.2. All files can be found in appendix G.

13.1 Abstract Syntax

A DTD formulates the syntax of a given XML document. The DST-L description is stored in an XML document. It is therefore possible to implement the abstract syntax of the DST-L by formulating a DTD describing the basic structure of an XML document containing a description. For the XML document containing the DST-L description of the running example, please refer to appendix J.2.3. The complete DTD can be found in appendix G.1.

The two elements network and tables each describes the syntax of the two parts of the DST-L.

In the specification the syntax of the network description is given by the four total functions returning data about the physical devices in the network. Therefore, the network element contains a points, a segments, a signals, and a sensors element.

```
<!ELEMENT network (points, segments, signals, sensors)>
```

The four child elements of the network element are similarly constructed, therefore only the segments element will be explained in details.

The segments function of the RSL has the following signature

value
  segments: SegmentId $\rightarrow$ (SensorId $\times$ SensorId $\times$ SegmentId-**set**)

In the DTD this is represented by

```
<!ELEMENT segments (segment*)>
```

Each segment element has three attributes representing the identifier of the segment and the identifiers of the two sensors defining the segment. Furthermore, a segment element has a crossing child element. If present, the crossing element contains a child element for each of the segments which crosses the parent segment element.

The SensorIds refers to an ID attribute of a sensor in the document, this is expressed by the IDREF. Furthermore, the SensorIds must be present in each segment element, this is expressed by defining the attributes as #REQUIRED.

```
<!ELEMENT segment (crossing?)>
<!ATTLIST segment
        segmentid ID #REQUIRED
        sensor1 IDREF #REQUIRED
        sensor2 IDREF #REQUIRED>
```

In the specification the syntax of the interlocking tables is given by the four total functions representing the actual interlocking tables. In the DTD this is implemented by a tables element having four child elements, one for each interlocking table.

```
<!ELEMENT tables (route_definition_table,
                  route_conflict_table,
                  point_position_table,
                  signal_setting_table)>
```

The four elements representing the interlocking tables are constructed in a similar way, therefore only the route_definition_table element will be explained in details.

The route_definition_table element represents the rdt of the RSL specification. In XML each possible pair of input and output data is represented by a tuple containing both. Therefore a function is represented by a list of tuples.

```
<!ELEMENT route_definition_table (route_definition*)>
```

The route_definition_table element has a list of route_definitions as child elements. These tuples represent the function of the RSL specification.

Each of the route_definition elements represents a single function call.

```
<!ELEMENT route_definition (route_sensor_list)>
<!ATTLIST route_definition
        RouteId ID #REQUIRED>
```

The route_definition element has an attribute representing the input value of the function. Additionally the element has a child element, route_sensor_list, containing a list of references to the sensors defining the route.

```
<!ELEMENT route_sensor_list (route_sensor*)>
```

13.2 Static Semantics

In the RSL specification the static semantics for the DST-L was formulated axiomatically. The axioms express properties of the observers in the model.

The static semantics of the DST-L is implemented by using the extensible stylesheet language (XSL) [W3Cc]. This language consists of three parts, two of which will be used - XSLT which is a query language for transforming XML documents and XPath which is a language for defining parts of an XML document.

An example of an XPath extracting all identifiers of points having the sensor G1 as turn sensor would look as follows:

```
description/network/points/point[@turn='G1']/@PointId
```

but can also be expressed as follows since the element point is only used at this location in the XML document:

```
//point[@turn='G1']/@PointId
```

In the following section the general translation techniques from RSL specification to XSLT is presented, and afterwards an implementation example is presented.

13.2.1 General Translation Techniques

When implementing the RSL specification using XSLT some general techniques have been used.

Universal Quantification ($\forall$) Universal quantification is often used in the RSL specification. In XSLT this is implemented by using the element for-each, looping over a set of objects defined by an XPath expression. If the quantification is performed on two objects of the same type, the second for-each element only loops over the remaining of the list containing the objects. This is to prevent examining any combinations of two objects twice.

Existential Quantification (∃) Existential quantification has mainly been implemented by finding all occurrences fulfilling the predicate, storing them in a variable element, and then counting the number of objects in the variable element. If the count is zero the existential quantification is not fulfilled. In a few cases another approach has been used (see [routes_crossing] and [entry_border]). In other cases a for-each element is used to express the existential quantification (see [routes_crossing]).

Conjunction of Predicates ($a \wedge b \wedge c$) Each predicate of a conjunction is evaluated separately by calling the relevant observer, and the result is stored in a variable element. The actual conjunction of predicates is evaluated by an if element where the condition is the conjunction of the values of the variables.

Cardinality (card) The cardinality is in XSLT calculated by using the function count. This function counts the number of nodes at a certain level in the XML tree defined by an XPath expression. An exception is the axiom [distinct_point_sensors] where the cardinality is performed by actually testing that the set contains the correct elements.

Set Membership (∈) The RSL set membership operator is in XSLT represented by an if element asking whether an object is in a set of objects, and returning true if it is the case.

If, Then, Else Expressions In XSLT the construction choose, when, otherwise corresponds to the RSL If, then, else expression.

Head and tail of a list (hd, tl) The functionality of the RSL hd is implemented by the XPath expression [1] returning the first element of a list of elements. The RSL tail is used to identify the head of the tail of a list. This is implemented using an index referring to the next element of the list.

Length of a list (len) The route_end axiom is implemented by using the XPath expression last() returning the last element of a list of elements. This implements the len functionality as it is used in the model of the DST-L, since the len is only used to find the last sensor in a sensor list.

In two axioms ([point_direction_of] and [ppt_entries]) a Nat is used in the specification to denote specific elements of lists to formulate the axiom. In the XSLT implementation the XPath function position() is used returning the position of the current element.

In some axioms it has been necessary to use a variable element containing the complete XML tree. This is because when using a temporary tree[1], it is not possible to search in the original XML tree unless it is stored in a variable. This is the the case in for example [conflicts_safety_with] and [conflicts_physical_with].

[1] A tree constructed at run time.

13.2.2 Implementing Observers and Axioms

The RSL specification of the static semantics is given by a number of axioms. An example of the implementation of these is presented in this section.

The model contains observers each returning information about concrete objects in the network. In the XSLT implementation each observer is represented by a named template. A named template can be called by its name with concrete parameters and is therefore similar to functions in RSL.

An example of an implementation of an observer is:

```
value
  border : SensorId → Bool
  border(s)≡card { seg | seg : SegmentId • s ∈ sensors_of(seg) } = 1
```

The observer border returns true if the input sensor is a border sensor.

The XSLT implementation counts the number of segments which the sensor is defining. If this number is one, true is returned.

```
<xsl:template name = "border">
  <xsl:param name = "sen"/>
  <xsl:if test="count($description//segment[@sensor1=$sen or @sensor2 =$sen])=1">
    true
  </xsl:if>
</xsl:template>
```

The axioms in the model representing the static semantics of the DST-L are implemented in XSLT as follows. For each axiom a named template tests that the property which the axiom was to exclude is not present in the network description or interlocking tables. If a property is present, the template returns an error message. This means that there is a slight difference between the axiom and the corresponding template. The axiom expresses what is wanted, the template checks if an unwanted construction is present and if so returns an error message. An example implementation of an axiom is presented:

```
axiom
  [distinct_stem] ∀ p1, p2 : PointId • p1 ≠ p2 ⇒ stem(p1) ≠ stem(p2),
```

The axioms assures that no two points have the same stem sensor.

```
<xsl:template name = "distinct_stem">
  <xsl:for-each select = "//point">
    <xsl:variable name = "point_pos" select = "position()"/>
    <xsl:variable name = "point1" select = "@PointId"/>
    <xsl:variable name = "stem1" select = "@stem"/>
    <xsl:for-each select = "//point[position() &gt; $point_pos]">
      <xsl:variable name = "point2" select = "@PointId"/>
      <xsl:variable name = "stem2" select = "@stem"/>
      <xsl:if test = "$stem1 = $stem2">
        <error>
          The two points <xsl:value-of
          select="$point1"/> and <xsl:value-of
          select="$point2"/> have same stem sensor.
        </error>
      </xsl:if>
    </xsl:for-each>
```

```
    </xsl:for-each>
  </xsl:template>
```

In the XSLT implementation two for-each elements loop over points, the first over all points and the second only loops over the remaining of the point list based on the position of the first point. For each of the points the stem is found by calling the template stem, and an if element tests whether the two stem sensors are equal. If this is the case an error is returned.

The implementation of the static semantics is stored in two files, one for the network description, well_formed_network.xsl, found in appendix G.2.1 and one for the interlocking tables, well_formed_tables.xsl, found in appendix G.2.2. All templates representing the axioms are called from a third file, static_semantics.xsl found in appendix G.2, which collects the error messages returned from the templates.

13.3 GUI for Collection of Data

In this section an application for collection of data for the DST-L is presented. The application is called "The Tramway Network Data Collector" (TNDC).

As described in the beginning of this chapter the data for the network description and the interlocking tables of a concrete tramway network is stored in an XML document. To avoid the necessity of users to write data directly in an XML document, an interface for collection of data and storing it in an XML document has been developed.

In this section the development of the TNDC is presented. For the source code implementing the application please refer to appendix G.3.1. In appendix P a user's guide to the TNDC can be found.

Developing the TNDC

The TNDC is developed using XForms [W3Ca]. XForms is a relatively new XML tool for creating net based GUIs. XForms specifications are divided into the XForms core module and the user interface. XForms is not a stand alone language, but requires a host language such as XHTML.

The XForms Core Module

The XForms core module contains the overall structural elements of the XForms specification. The module includes the following elements: the model element with the child elements: instance, submission and bind.

The model element. The model element is the root of the core module. It may have an attribute schema which identifies one or more XML Schemas which the collected data must be validated against.

The data collected in the TNDC must comply with the abstract syntax of the DST-L. In the implementation of the DST-L the abstract syntax was represented by a DTD. XForms only support XML Schemas and therefore an XML Schema has been developed expressing exactly the same as the DTD formulating the abstract syntax. Please refer to appendix G.3.3 for the complete XML Schema.

```
<xforms:model schema = "dstl_schema.xsd">
  <xforms:instance> ... </xforms:instance>
  <xforms:submission> ... </xforms:submission>
  <xforms:bind> ... </xforms:bind>
</xforms:model>
```

The instance element. The instance element contains or references the instance data.

The XForms specification is initialized based on instance data. The instance data is represented by an XML document containing the description data of a specific network. The initial instance data contains the minimum skeleton of the resulting XML document. The XML file containing the initial instance data can be found in appendix G.3.2.

```
<xforms:instance src = "description.xml"/>
```

The submission element. The submission element contains information about what to submit and how to submit it.

In the implementation of the TNDC data is submitted to the file containing the instance data. This way the next time the TNDC is loaded, it contains the latest submitted data.

The implemented TNDC does not facilitate the user to select arbitrary instance data. This means that each application only support a single specific tramway network.

```
<xforms:submission action = "description.xml"
                   method = "put" id = "submit" replace = "none"/>
```

Bindings The bind element is optional. Each binding binds a node set to an identifier. In the implementation of the TNDC no bindings are used.

The User Interface

The GUI of the TNDC is constructed mainly by using the functionality of XForms for designing GUI's. The GUI is divided into two sections, one for the network description, collecting data for the sensors, segments, points, and signals and one for the interlocking tables, collecting data for the interlocking tables.

Each section of the GUI contains four buttons, one for each kind of device and one for each interlocking table. Each button is connected to a trigger, by clicking a button the corresponding trigger is activated. The trigger is related to a switch construction. When activating the trigger, the case of the switch changes. Each switch has two cases one for showing and one for hiding the data of the corresponding device or table. The implementation is quite similar for all, and will only be explained in details for segments.

Furthermore a button Write To Disk activates the submit functionality as specified in the core model. This results in the data keyed in to the forms by the user is submitted to the XML document containing the instance data.

Displaying Data Initially all data is hidden. The switch structure called SegmentHider has two cases called segment_hide and segment_show. In case segment_hide only the button Show Segment Details is visible. The button is related to a trigger, and when activating the trigger by pressing the button, the state changes to segment_show. This results in the data, describing the segments, becomes visible.

```
<xforms:switch id = "SegmentHider">
  <xforms:case id = "segment_hide">
   ...
  </xforms:case>
  <xforms:case id = "segment_show">
   ...
  </xforms:case>
</xforms:switch>
```

The GUI showing the data is constructed by using the repeat functionality of the XForms which loops over a given node set. An XPath expression identifies the node set containing all segments. The segmentidx identifies the currently active segment element in the node set.

```
<xforms:repeat id="segmentidx" nodeset="/description/network/segments/segment">
   ...
</xforms:repeat>
```

For each segment a number of input fields, corresponding to each attribute and child element of the segment element, are added to the GUI.

```
<xforms:input ref="@sensor1" class="datainput2">
  <xforms:label class="label_text">Sensor 1</xforms:label>
</xforms:input>
```

The attribute ref identifies the instance of the XML document which the input represents, and the class attribute is used by a cascading style sheet, that defines properties of the GUI such as fonts (please refer to appendix G.3.4).

To be able to display several crossing segments for a single segment element an additional repeat structure is nested inside the first, showing all crossing segments corresponding to the considered segment element.

Additionally a trigger Add Crossing Segment is added which inserts a new crossing_segment element in the tree structure of the considered segment element.

```
<xforms:trigger>
  <xforms:label class="label_text">Add Crossing Segment</xforms:label>
  <xforms:action ev:event="DOMActivate">
    <xforms:insert at="index('crossingidx')"
                   nodeset="/description/network/segments/segment[index('segmentidx')]/
                   crossing/crossing_segment" position="after"/>
  </xforms:action>
</xforms:trigger>
```

Also a trigger Remove Crossing Segment is added to facilitate removing a crossing segment from the XML tree structure. The user fills out the forms with data describing the segments in a concrete network.

Corresponding to the triggers for crossing segments two triggers Add Segment and Remove Current Segment is added to the GUI, respectively adding and removing segment elements to and from the XML tree structure. A third trigger Hide Segment Data activates the switch and causes a case change back to segment_hide.

13.4 Graphical Representation of the DST-L

To present the information written in the DST-L a graphical representation has been developed. This was done using XSLT and HTML.

The network description is presented as four tables containing data about sensors, segments, points and signals.

The interlocking tables are presented as the tables they are.

The source code and the graphical representation corresponding to the running example can be found in appendix G.4.

13.5 Chapter Summary

The implementation of the abstract syntax of the DST-L in a DTD and the static semantics using XSLT has been presented. Furthermore an application for collection of data and a graphical representation of the network description and the interlocking tables has been presented.

Chapter 14

Implementing the STS-L

In this chapter the implementation of the model of the State Transition System Language (STS-L) and the control algorithm of the control system are presented.

The implementation is performed in the imperative language C [KR88], and hence this chapter deals with the translation of the RSL models to C programs.

The overall structure of the implementation is presented in section 14.1.

The implemented models are the *abstract syntax*, the final refinement of the *imperative operational semantics*, and the *control algorithm (interpreter)*. The implementation of these are described in sections 14.2, 14.3, and 14.5 respectively.

Please note that this means that the static semantics of the STS-L is not implemented. This is quite natural, since the treated STSs, rules, conditions, and actions in the implementation all originate from the generator and are therefore assumed to be well-formed, as argued at model level in section 8.4. In practice the static semantics could have been implemented and the STSs could have been checked, but this has not been done, as the STSs has been found to be statically well-formed in the model and the implementation follows the model.

The chapter deals with the main differences of model and implementation. Therefore, only part of the implementation is presented. For the entire program please refer to appendix H.

14.1 Program Structure

The implementation consists of several C programs. The overall structure of inheritance among the program files is illustrated in figure 14.1. A normal arrow is inclusion of the source file, while a dashed arrow means that some functions are declared externally in the target file to make them accessible. The dotted squares around program parts means that they could have been made in one part, but are kept separate to add clarity to the program structure.

module : The *module* part is the files used for scenarios, testing, and simulation with seperate makefiles compiling the programs. The different modules are commented in their respective chapters being 16 for scenarios and testing, and 17 for simulation.

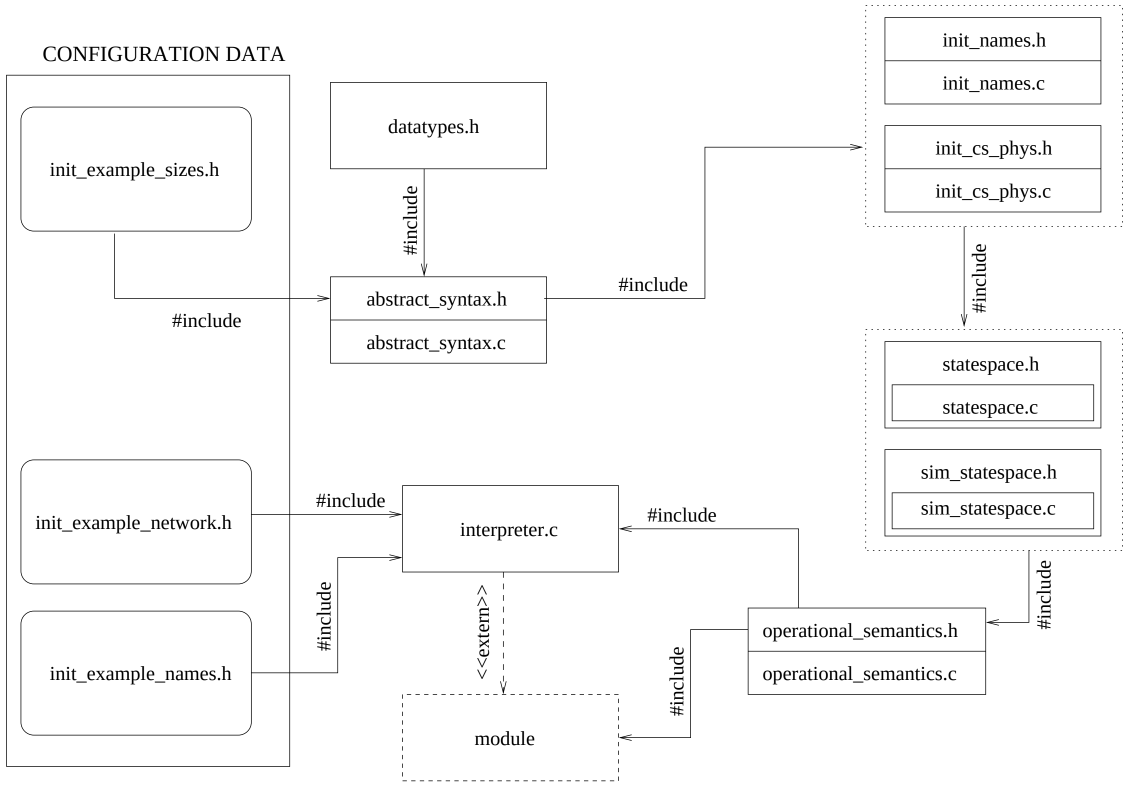

Figure 14.1: Structure of implementation of STS-L

The box named *CONFIGURATION DATA* refers to the data generated[1] from a description in the DST-L. The rest of the program code is generic. The configuration data consists of three header files:

- init_example_network: This header file contains the STS control components and STSs of the physical environment generated from the description in the DST-L, as well as the initial state space.
- init_example_sizes: As opposed to RSL where lists can be defined without any parameter of the size, arrays in C must be initialized to a size. This header file contains sizes for the arrays for conditions, actions, and rules. Furthermore, it contains the number of different devices used mainly to initialize the different parts of the state space in the init_cs_phys program.
- init_example_names: This file contains auxiliary functionality to facilitate printing of events in the system. The different identifiers are represented as strings in the XML representation which translates to integers in the C program, used as indexes in the arrays. To facilitate the printing of the original string names, they are stored in this file, and the names can be retrieved by looking up with the C identifier.

The datatypes header file contains basic data types used throughout the program, including the different words (double word, quad word, etc.).

The init_names and init_cs_phys files found in appendix H.5 initialize the arrays used for names and STSs, respectively.

The statespace and simulation_statespace[2] files found in appendix H.3 contain the different

[1]The implementation of the generator is presented in chapter 15
[2]Denoted *sim_statespace* in the figure

arrays of the state space as well as functions for retrieving information from and updating them. The actual arrays have been encapsulated so that they can only by accessed through the functions. This is illustrated in the figure by putting the square representing the *.c* file completely within the square representing the header file.

14.2 Abstract Syntax

The implemented abstract syntax of the STS-L is presented in this section. Please refer to appendix H.1 for the source code.

The basic types of the abstract syntax (in STS_abstract_syntax.rsl) are implemented by struct declarations. Thus, for Sts, Rule, Condition, and Action a struct is declared. This corresponds well to the type declaration in the model. Additionally, all actions, conditions, and rules are stored in three arrays (one for each) in the implementation, as opposed to the model where conditions and actions are stored in the rules, and rules in the STS.

For instance, a condition-list of a rule in the model translates to a start index for the conditions and a number of conditions in the struct, referring to a part of the global condition array of the implementation. The lists are implemented this way for Rules and Sts, meaning the structs has start index and a number. Thus, the Rule struct becomes;

```
typedef struct{
  uint32_t no_cond;
  uint32_t start_idx_cond;
  uint32_t no_act;
  uint32_t start_idx_act;
  } Rule;
```

Where uint32_t is an unsigned 32 bit word (double word).

The Condition struct consists of a number of parts corresponding to the RSL type. body is an extension of the Relation part of the RSL containing both the relation and a value itsConstant

```
typedef struct{
  uint32_t idcomp;
  uint8_t body;
  uint64_t expr;
  } Condition;
```

Thus, the constructor is implemented as;

```
void mk_Condition(Condition *cond
                  uint32_t objId,
                  IdCompCodes location,
                  Relation relation,
                  uint8_t itsConstant,
                  uint64_t expr)
    {
      cond -> idcomp = mk_Idcomp(objId, location);
      cond -> body = (itsConstant << 6) | ((uint8_t)relation & 0x3F);
      cond -> expr = expr;
    }
```

Where itsConstant maintains information on whether the expr word should be treated as an IdComp (istConstant equal to zero), AddIdComp (istConstant equal to two), or a State word (istConstant equal to one). The expr word is a 64 bit word (quad word), because if it is an AddIdComps word it will contain two IdComp words.

Similarly an Action struct has been implemented.

IdCompCodes, the location part of the IdComp, Operation, Relation, and the different states (e. g. PointState) are all implemented as enumerations, making the maximal type a double word. The information in the types are placed in the word, for instance relations translate to the numbers from 0 to 4. This differs from the model in the aspect that the enumerations have an actual Natural value which is not the case in the variant definition enumerations of the model.

Since the IdComp type indirectly through the variant definition consists of an identifier part and a location part, the 16 least significant bit (low word) of the IdComp word contain the identifier, and the 16 most significant bit (high word) contain the location. Consequently, the function corresponding to the RSL IdComp constructor is implemented as:

```
uint32_t mk_IdComp{uint32_t objid, IdCompCodes location}
  {
    return (location << 16) | (objid & 0xFFFF);
  }
```

14.3 Operational Semantics

The implementation of the model of the *operational semantics* in the scheme l2_operational_semantics is straightforward and is presented in this section. Please refer to appendix H.2 for the source code.

For every value declaration in the RSL model a function is implemented. Due to the modular structure of the models, where a function exists for every manipulation or retrieval of data, the implementation of interpret_sts, interpret_rule, evaluate_condition_list, evaluate_condition, eval_idcomp, eval_addidcomps, eval_expr, process_action_list, and process_action are almost identical to the imperative model. "If else" statements and loops (while and for) are almost identical, and RSL case structures are implemented as the very similar C switch structures.

The implementation of the basic retrieval and update functions is almost identical to the model, as these functions only call functions of the state space.

Please note that the functions for updating the state space has an extra parameter used in simulation and testing deciding whether the values of changed variables in the state space should be printed to the screen.

Control structures have been added preventing the usage of index less than zero and out of bounds.

14.4 State Space

The implementation of the state space is presented in this section. Please refer to appendix H.3 for the source code.

The object arrays in the model are declared as arrays in the implementation. Likewise each function within the different objects is declared. Variables in the model that are hidden and, therefore, not directly accessible, are declared in a *.c* file and not declared externally in the corresponding *.h* file. Hence, they are not accessible by the rest of the program. The functions of the object array, for manipulating the hidden variables from the model, are declared externally in the implementation, making them available to be used in the *operational semantics*. See section 14.1 for the general program structure. In this way encapsulation of the state space is ensured.

The implemented arrays and their functions differ somewhat from the model. In the model variables contained the state of the objects. In the implementation 32 bit words (double word) have been used for objects with one variable, and 64 bit words (quad word) have been used for objects with two variables. In this way each state is maintained by the value of a 32 bit word. For instance, a point has a requested state and an actual state. The first 32 bit contain information about the actual state, and the last 32 bit about the requested state. Thus, initializing a point object can be done by the auxiliary function put_point_object:

```
void put_point_object(int i,
                      PointState actual_pointstate,
                      PointState requested_pointstate)
  {
    POI[i] = ((uint64_t)requested_pointstate << 32) |
             ((uint64_t)actual_pointstate & 0xFFFFFFFF);
  }
```

This function is not available in the model, but makes it easier to initialize the objects. Basically, the requested pointstate is put into the quad word and shifted 32 bit left, and the actual pointstate is then put in the word without modifying the other 32 bit.

Similarly, the retrieval and modification functions of the object arrays are implemented using bit shifting and masking. For instance the set_actual_pointstate_var is implemented as:

```
void set_actual_pointstate_var(int i, PointState pointstate, bool_t print)
  {
    if (i < 0 )
     { printf("index %d is less than zero \n", i);}
    else if (i >= NO_POINTS)
     { printf("index %d out of bounds for point array \n", i); }
    else if ((int)pointstate < 0 || pointstate > 3 )
     { printf("not valid actual pointstate %d \n", pointstate);        }
    else
     {
       POI[i] = ((POI[i] & 0xFFFFFFFF00000000) | ((uint64_t)pointstate));
       if(print) { print_point_object(i); }
     }
  }
```

Control structures have been added because the implemented data types are broader in the implementation of the state space and the principle can be deducted from the ”if else” structure of above presented function.

For instance the maximal type of *PointState* is *State* in the model, while in the implementation the maximal type is a 32 bit word. To prevent that variables of type *PointState* assume values that do not correspond to a valid pointstate a control structure has been implemented, returning an error message if such an assignment of values is attempted. Similarly, control structures have been added for the other types of states. Furthermore, control structures have been added, preventing the usage of index less than zero or out of bounds.

14.5 Control Algorithm

The model of the control algorithm in scheme `interpreter` has also been implemented. Please refer to appendix H.4 for the source code.

The only purpose of the interpreter is to call functions of the operational semantics on parts of the state space in a specific order. Thus, both the model and the implementation consist of loops calling other functions, and therefore they are almost identical. Please note that the interpreter does not initialize the state space as in the model. This is done in the beginning of the main programs of the different modules.

14.6 Chapter Summary

The implementation of the abstract syntax and the operational semantics of the STS-L in C has been presented. Additionally, the implementation of the state space has been presented.

Chapter 15

Implementing the Generator

In this chapter the implementation of the generator is presented.

The purpose of the generator is to generate the configuration data used to initialize the C implementation of the control system.

The generation is based on a description of a specific tramway network and the interlocking tables formulated in the DST-L. The implementation of the DST-L is done in XML. A force of the XML language is the transformation of XML documents into new formats using XSLT [W3Cc], and therefore, XSLT has been chosen to implement the generator.

The implementation of the generator is based on the RSL model I_GENERATOR presented in section 11.3. This chapter deals with the translation of the RSL specification to XSLT.

In section 15.1 the structure of the implementation is presented, and in section 15.2 the process of generating the configuration data is explained.

For the entire implementation please refer to appendix I

15.1 Program Structure

The structure of the implementation of the generator consists of four XSL transformations. One generating an XML document containing all necessary data, extracted from the description of the network in the DST-L, and three XSL transformations, each generating one of the three header files. Figure 15.1 illustrates the overall structure of the generator implementation.

In the figure an arrow marked *«input»* identifies the input XML file for an XSLT, similarly an arrow marked *«output»* identifies the output of the XSLT. If an XSLT needs data from additional XML documents, an arrow marked *«relation»* illustrates the relationship. An XSLT can call templates in other XSLT documents by inclusion. An arrow marked *«include»* illustrates an inclusion.

15.2 Generating the Configuration Data

The purpose of the generator is to produce the configuration data for the C program described in chapter 14. The configuration data constitutes three header files: init_example_network, init_example_sizes and init_example_names.

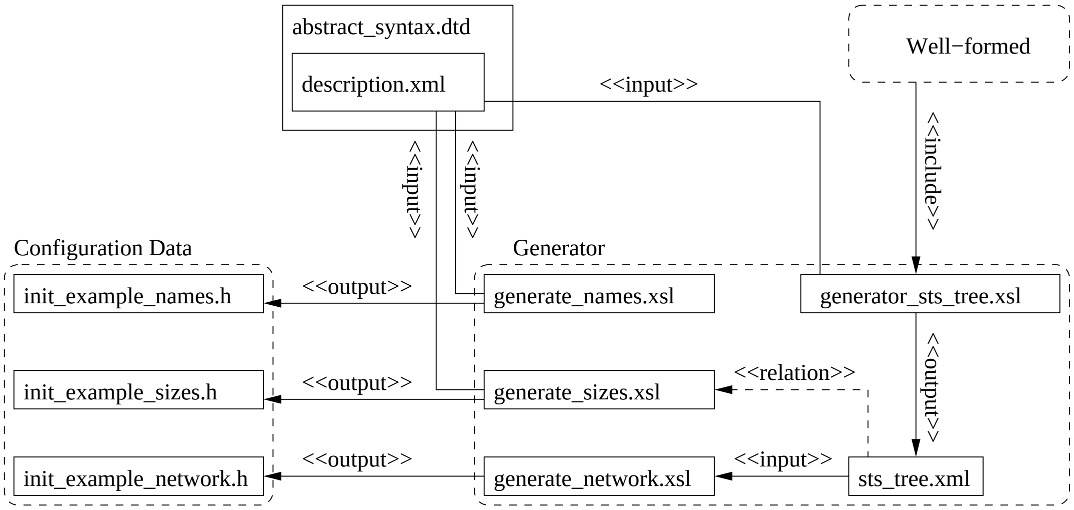

Figure 15.1: Structure of XSLT implementation of the generator

15.2.1 init_example_network

This header file contains the instantiation of the state space, the STS control components, and the STSs representing the physical environment.

State Space The instantiation of the state space is performed by instantiating each of the arrays which constitutes the state space in the C implementation. The initial values corresponds to the presentation in section 8.1. For the array containing the sensors, the instantiation is performed as follows:

```
for (sensorIdx = 0 ; sensorIdx &lt ; NO_SENSORS ; sensorIdx ++ )
{
    put_sensor_object(sensorIdx , S_LOW );
}
```

where the constant NO_SENSORS is initialized in init_example_sizes.

The generation of the initial state space is not dependant on the description of the network. The reason why it is part of the generator and not the C program is, that by including it in the generator it is possible to construct other initial state spaces possibly depending on the description in the DST-L.

Identifier Translator In the RSL model of the generator a number of functions is specified translating identifiers from the model of the DST-L to the corresponding identifiers in the model of the STS-L. The functions were only given by function signatures since a more concrete description of the form of the identifiers was not intended.

In the implementation of the generator these functions are represented by indexes. For the sensorid_to_stslsensorid a corresponding index sensor_index contains the DST-L object identifiers and the STS-L object identifiers. In the DST-L objects are identified by a string. In the STS-L objects are identified by the index where data about the object is found in the array of the considered object type.

```
<xsl:variable name="sensor_index">
  <xsl:for-each select="//sensor">
    <xsl:variable name="sensor" select="@SensorId"/>
    <sensor>
      <xsl:attribute name="SensorId">
        <xsl:value-of select="$sensor"/>
      </xsl:attribute>
      <xsl:attribute name="index">
        <xsl:value-of select="position()-1"/>
      </xsl:attribute>
    </sensor>
  </xsl:for-each>
</xsl:variable>
```

Generation of STSs The generator generates STSs representing the control components and the physical environment. The generation is performed in two steps. The first step collects the data to express the STSs. And the second produces the actual header file to be used by the C-program.

The first step results in an XML document (sts_tree.xml) which contains an element for each necessary STS. Each STS element contains a number of rule elements. Each rule element contains a conditions element and an actions element. The conditions element contains a condition element for each condition of the considered rule, and actions element contains an action element for each action of the rule. The condition and action elements are constructed by two seperate templates. The complete XML document is constructed based on the structure expressed by the model of the generator (GENERATOR.rsl).

```
<sts>
  <rule>
    <conditions>
      <condition>
       ...
      </condition>
    </conditions>
    <actions>
      <action>
       ...
      </action>
    </actions>
  </rule>
   ...
</sts>
```

The second step generates the actual C code. Based on the XML document containing data for each STS, the header file init_example_network is generated by extracting the data from the XML document.

The XLST generating the XML document can be found in appendix I.1. The XSLT generating the actual C code can be found in appendix I.2.

The generated XML tree for the running example network can be found in J.2.4. and the generated instantiated control components for the running example network can be found in appendix J.2.5.

15.2.2 init_example_sizes

This header file contains the C code initializing a number of constants based on the description of the network formulated in the DST-L. From the description of the network it is easy to count the number of the different types of devices. To decide the number of rules, conditions and actions, the count is performed on the XML tree generated in the first step of the generation of the network header file. The source code for the XSLT generating the sizes is found in appendix I.3.

The header file for the running example network can be found in appendix J.2.6

15.2.3 init_example_names

This header file only contains data used for printing purposes. The data is generated simply by iterating over each object generating a string which is printed to the output file. The source code for the XSLT is found in appendix I.4·

The header file for the running example network can be found in appendix J.2.7

15.3 Chapter Summary

The generation of configuration data for a control system from a description in the DST-L using XSLT has been presented. Furthermore the implementation of the generator is described.

Chapter 16

Test

In this chapter the testing of the implementation of the DST-L (see chapter 13), the STS-L (see chapter 14) and the generator (see chapter 15) is presented.

In the first section the approach to testing in general is presented, and in the following sections the structural and functional tests (for an introduction see [GJM03]) conducted on the above mentioned implementations are presented.

The running example network and a second network, displaying properties of the domain not present in the running example network, have been instantiated for the testing. The instantiation data, including graphical representation in DST-L, XML DST-L description, XML STS-tree, and C header files, can be found in appendix J.2 for the running example network and in appendix J.3 for the second network.

16.1 Testing in General

Testing is done to discover errors. A test in which an error is discovered and corrected, will in most cases ensure that the program has one less error. A thorough testing in which no errors are discovered does not guarantee that no errors exist in the program, it just increases the confidence in the correctness of the code.

Even with a complete verification of the models (see chapter 10), an error may be introduced during implementation. To minimise this risk the models have been developed to be close to the implementation. Nevertheless, this is not a guarantee that the programs do not contain any errors, and therefore testing is necessary to create confidence in the implementation.

16.2 Structural Tests

Structural testing, also called ”white box” testing, consists of test cases based on the internal structure of the program, hence the name. The general tactic is to construct a test case for every distinct part of the code. For instance an ”if” condition would require at least two test cases, one satisfying the condition and one not satisfying it. Depending on the specific condition several test cases would be necessary to test all possible ways of satisfying and not satisfying the condition.

The templates representing the observers of the RSL model have been tested in a structural test. For each template corresponding to the observers returning concrete data about objects, test cases has been constructed to show that the correct data is returned. The test of the templates corresponding to the derived observers is not exhaustive. The test cases which could be constructed based on the description of the running example have been constructed and the templates have been tested.

The test program is an XSLT calling the templates with concrete parameters. For the source code of the test program, please refer to appendix L.1. The results of the structural test is presented M.2.

The implemented functionality of the operational semantics has been tested in a structural test. The test cases at the lowest level of the functionality have been found by unfolding the functionality called from the state space. Thus, implicitly the functionality of the state space has also been tested this way. The structural test is not complete in the sense that not all possible test cases have been generated. For instance certain "if" conditions in the implementation consist of a logical expression consisting of up to four simple expressions connected by the `or` operator. For a complete structural testing of entering this part of the code it is not sufficient with one test case. 5 test cases are needed to test all possible ways of entering the code, as C uses conditional logic. In such an instance only one or two of these test cases have been generated. Furthermore, the test cases have not been generated by any special technique, but by visually inspecting the code. Even though the inspection has been systematic this may lead to relevant test cases that have been overlooked.

The module used in testing the C programs consists of the `test.c` program which can be found in appendix N.1.

For the test cases their results and expected results, please refer to appendix O.2.

16.3 Functional Tests

Functional testing, also called ”black box” testing, only focuses on the input and output of the functions, without caring about the internal structure of the program. Functional testing is therefore a test of whether the program does ”what it is supposed to do”.

The functional testing is divided into two parts, one testing the DST-L and one testing the STS-L and the generator.

The functional testing conducted on the DST-L concerns testing of the static semantics. For each of the templates representing axioms a description of a network which does not comply with the condition is formulated and tested. The result of this test must produce an error corresponding to the considered axiom. A network description which is known to be well-formed is then tested against the considered condition. This test must produce no errors. By performing this test of the DST-L the confidence in its correctness is increased.

For the test results of the functional tests performed on the DST-L please refer to appendix M.3.

The second part of the functional testing conducted is scenarios of the system, thereby testing the operational semantics and the state space functionality of the STS-L, as well as the generator, since the scenarios use the generated data. Scenarios are created for traversal of each of the routes from the instantiated running example and the second instantiated network. Furthermore, scenarios have been created to illustrate the interaction between conflicting and non-conflicting routes.

The scenarios consist of the setting of certain variables in the state space and the calling of the interpret_sts functionality on different STS components corresponding to the traversal of one or more routes.

The module used in running the scenarios of the running example network consists of a header file containing actual scenario descriptions, named routes.h, which can be found in appendix N.2.3, and a main program running through the scenarios, named scenario.c, which can be found in appendix N.2.5. The compilation of the STS-L implementation with this scenario module is handled by the makefile scenario_run which can be found in appendix N.2.1. The module is illustrated in figure 16.1.

Similarly, the module for the second instantiated network consists of the files routes2.h (appendix N.2.4), scenario2.c (N.2.6), and the makefile scenario_run2 (N.2.1).

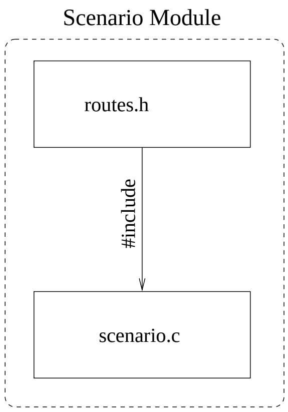

Figure 16.1: C scenario module

With the scenarios it has been inspected that the control components behave as intended, even if other scenarios concerning the same components have already been run. In this way the confidence in the correctness of the STS components generated is increased, and as a consequence also the confidence in the correctness of the generator is increased. Furthermore, the implementation of the STS-L is tested in addition to the structural testing, as the functionality is used in the scenarios. Thus, the confidence in this implementation is further increased.

Please note that even though the generated data behaves correctly when manipulated by the implemented STS-L, and this indicates that the generator is behaving correctly, it is by no means an insurance that the generated data will be correct for all instantiated examples. For a more thorough testing of the generator additional descriptions in the DST-L should be instantiated, and the corresponding control components and physical environment should be generated and tested.

For the test scenarios their results and expected results please refer to appendix O.3, with results for the running example network in O.3.1 and for the second network in O.3.2.

16.4 Test Results

Both the structural and functional tests have been successful, yielding the expected results. Thus, the confidence in the implementation of the DST-L, the STS-L and the generator has been increased.

For more specific comments on the test results please refer to appendices M.1 and O.1

16.5 Chapter Summary

In this chapter the concepts of structural and functional testing have been introduced. Furthermore, the testing of the implemented models is presented. This includes structural and functional testing of the implemented STS-L, functional testing of the implemented DST-L, and to some extent also functional testing of the generator.

Chapter 17

Simulation of Control System

In this chapter a simulation of a working control system is presented based on the interpreter and the STSs simulating the physical environment. In section 17.1 the C simulation module is presented, in section 17.2 the simulation of trams in the system is presented, in section 17.3 the simulation of route traversals is presented, in section 17.4 the simulation of the physical environment is presented, in section 17.5 the simulation of the concurrent behaviour of the environment is presented, in section 17.6 the synchronisation of the parallel processes is presented, and finally in section 17.7 different problems of the implemented simulation is commented on.

The simulation uses the instantiated C header files of the running example which can be found in appendices J.2.5, J.2.6, and J.2.7.

The simulation programs can be found in appendix K.

17.1 C-Module

The module used in the simulation is illustrated in figure 17.1. It consists of:

- A makefile thread_run (appendix K.1) handling the compilation of the implemented STS-L with the simulation module.
- A program, named phys_simul.c (appendix K.2), consisting of functions simulating each device by interpreting its STS in a non-terminating while-loop, and functions simulating the traversal of routes.
- A header file and program, named tram_event.h and tram_event.c (appendix K.3), initializing the tram event variables[1].
- A program, named thread_simular.c (appendix K.4), maintaining the parallel running of the interpreter, the different physical devices, and the simulation of route traversals.

17.2 Tram Events

The STSs simulating the sensors express when a tram can arrive and consequently when the sensor can become *high*. Thus, in the simulation sensors will become *high* whenever it is

[1]See section 17.2.

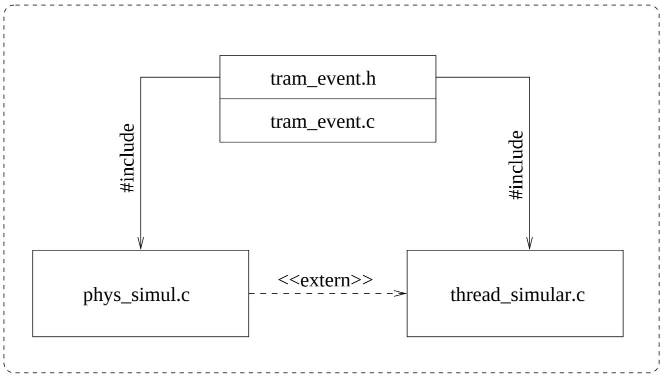

Figure 17.1: C simulation module

possible for a tram to arrive, which is not desirable. Therefore, it is necessary to simulate tram events.

This has been done by introducing a variable tram_events[i] for each sensor. If a tram has just passed the sensor with its front end, the variable is one and otherwise it is zero. These are also the variables that are used to simulate the traversal of a route in the different route threads.

17.3 Route Simulation

The traversal of routes is simulated by programs setting route request and tram event variables. Below part of the program for the traversal of route 1 of the running example network is presented. The sleep(i) function makes the program wait for i seconds. Thus, the program below can be summarised:

1. The variable tram_events[0] is set to one simulating that a tram passes sensor *G20.0*.
2. The program wait for 3 seconds simulating the time before the tram driver requests the route.
3. The route request variable is set.
4. The route simulation loops until the entry signal shows *go*
5. The variable tram_events[1] is set to one simulating that a tram passes sensor *G20.1*.
6. The rest of the program is not included, but the tram event variables are set with 3 seconds interval to simulating the tram driving through the route.

```
void route1(void)
{

  printf(" \n");
  printf("------------------------------------------\n");
  printf("Tram motion in front of G20.0 \n");
  tram_events[0] = 1;
  printf("------------------------------------------\n");
  printf(" \n");

  sleep(3);

  printf(" \n");
  printf("------------------------------------------\n");
```

```
    printf("A tram requests route 1 \n");
    set_route_requested_state(0, _true, 1);
    printf("————————————————————————————————\n");
    printf(" \n");

    SignalState start_signal_showing = get_actual_signal_state(0);

    while (start_signal_showing == S_STOP)
    {
      sleep(1);
      start_signal_showing = get_actual_signal_state(0);
    }

    printf(" \n");
    printf("————————————————————————————————\n");
    printf("Tram motion in front of G20.1 \n");
    tram_events[1] = 1;
    printf("————————————————————————————————\n");
    printf(" \n");


    .......


}
```

17.4 Simulation of Physical Environment

Each device is simulated by a program consisting of a non-terminating while-loop. In this loop the device STS is interpreted at each step. For sensors extra constraints have been added in the form of an "if then else" construction. These constraints corresponds to the may_interpret_sensor predicate and ensures that the timing assumptions are not violated in the simulation. Furthermore, an extra constraint ensures that when the sensor is *low* it may only be interpreted if the tram event variable is set to one. Below part of the program for sensor *G20.2* is presented:

```
void sensorG202(void)
 {
   while(1)
    {
     /* States of sensors, and the counters and control counters */
     uint32_t sG202 = get_sensor_state(2);
     uint32_t cG201 = get_counter_sen(1);
     ....
     uint32_t ccG201 = get_control_counter_state(1);
     ....

     /* CMV state for the counter controller */
     uint32_t cmvccG202 = get_cmv_state(9);

     /* Assumptions are implemented as extra conditions */
     if (sG202 == S_HIGH && (cG202-(cG201-cG203)) == (ccG202-(ccG201-ccG203)) &&
        (cG202-(cG210-cG242)) == (ccG202-(ccG210-ccG242)))
```

```
        {
         interpret_sts(sensor_simul[2], 1);
        }
       else if (sG202 == S_LOW && tram_events[2] && cmvccG202 == READY_TO_COUNT)
        {
         interpret_sts(sensor_simul[2], 1);
         tram_events[2] = 0;
        }
      }
  }
```

17.5 Concurrency

The concurrency of the environment is simulated using POSIX[2] threads. A thread[3] is initiated for each device, the interpreter, and for each simulated route (all in all this amounts to 26 threads in this simulation) - making the simulator a multi-threaded program. These threads share the global variables of the state space and run in parallel using time slices of the CPU. Part of the main program is presented below, with only the thread for the interpreter and for sensor *G20.2* included, this way exemplifying the creation of threads without including all 26:

```
int main()
 {
   pthread_t tid[26];

   initmain(); /* the generated STSs */
   initnames(); /* the generated names of devices */
   settramevents(); /* variables simulating tram events */

   if ( pthread_create(&tid[0], NULL, (void *)interpreter, NULL) )
    {
     printf("error for interpreter thread - exit \n");
     exit(10);
    }
   .......

   if ( pthread_create(&tid[3], NULL, (void *)sensorG202, NULL) )
    {
     printf("error for sensor G20.2 thread - exit \n");
     exit(10);
    }
   ......

   while (1)
    {
     sleep(1);
    }

   return 0;
 }
```

[2]Standard for C programs.
[3]Also known as light-weight process.

The function pthread_create creates the thread and returns zero if successful which is why the ”if” statement is included. The third parameter is the function running in the thread.

17.6 Synchronisation

The threads read and write common variables of the global state space. As mentioned in section 10.3.4, the parallel execution might actually cause the actions of a rule to be processed, despite the conditions not being fulfilled. Therefore, certain possible interleavings of the program are undesirable and should be avoided by some kind of synchronisation mechanism. Thus, mutual exclusion[4] of the critical sections[5] should be ensured. These undesirable interleavings corresponds to the temporal properties of the devices considered in section 10.4. Thus, these interleavings are not present in the ”real world”, but because time is not considered, they must be avoided by synchronisation.

Basically, as mentioned in section 10.3.4 the interleavings that are undesirable are of the form:

- rule 1's conditions evaluate to true.
- rule 2's conditions evaluate to true.
- rule 2's actions are processed.
- rule 1's actions are processed.

If the conditions of rule 1 is dependent of the variables written by the actions of rule 2 synchronisation is necessary.

17.6.1 General Synchronisation

The control component STSs and the STSs simulating the physical environment have an inherit synchronisation in the specific construction of conditions and actions. For instance a route controller and a point[6] read and write common variables, but they cannot fire rules at the same time as their conditions logically contradict each other[7]. This is true for many pairs of STSs, but some of the STSs running in parallel that read/write common variables can fire rules at the same time. For instance a sensor that is *high* and the corresponding counter controller that is *ready to count* can fire a rule at the same time. Nevertheless, as argued in section 10.4 this will not occur in the real world due to timing conditions.

To simulate this the synchronisation is achieved by adding extra conditions in the loops simulating the sensor devices as described in section 17.4, thereby, ensuring that only one thread enters a critical section at a time. These conditions corresponds to the may_interpret_sensor predicate, which can be found in appendix C.1 and was commented on in section 10.4.

Assuming correctness and completeness of the may_interpret_sensor predicate, the conditions generated from it in C, and the informal argumentation of the safety of the system presented in appendix D, this will ensure mutual exclusion of the critical sections.

[4] At most one thread at a time is executing its critical section.
[5] Section where the thread reads and/or writes variables used by other threads.
[6] Obviously this is only relevant if the point is in the route of the route controller.
[7] For a specific argument why please refer to appendix D.4.3

17.6.2 Output

The output of the simulator is an event based printing of variable values in the state space to the screen. Thus, the values of the variables are printed when they are modified by an action. A printing example is presented below. The example concerns the events that will occur when a tram passes sensor G21.0:

sensor	G21.0	S_HIGH	physical sensor state
counter	G21.0	2	sensor counter
cmv	G21.0	SENSOR_HIGH	cmv for sensor simulation
control counter	ccG21.0	1	control counter for sensor
cmv	ccG21.0	COUNTED	cmv for counter controller
sensor	G21.0	S_LOW	physical sensor state
cmv	G21.0	SENSOR_LOW	cmv for sensor simulation
cmv	ccG21.0	READY_TO_COUNT	cmv for counter controller

To ensure that variables are printed in the order they are modified by processing actions synchronisation is needed. This is done by making the process_action function atomic using a mutex lock[8]. This lock ensures that only one thread can process and print an action at a given time. This is a somewhat brute force approach to the problem and may slow down the system as even two completely independent STSs cannot process an action simultaneously. Nevertheless, it is necessary to ensure that the output of the simulator corresponds to the actual interleaving of the processed actions (events) of the STSs.

17.7 Traps & Pitfalls

Certain aspects of the simulation are error prone. This is mainly due to shortcomings of the simulation of tram events and the fact that the threads is not actually a case of true concurrency.

The tram event variables are unfortunately not a good simulation of the real world. In the real world when the trams front end pass the sensor, the sensor will instantly notice this and go to state high. With the simulated tram events the thread has to notice that the variable has been set and depending on the specific interleaving of events this may take a little time. Thus, the detection of trams by the sensors is not instant in the simulation.

Furthermore, the tram event variables are set to one in the appropriate route thread and to zero in the thread of the appropriate sensor. There is no mechanism ensuring when tram events are set. Thus, tram events of a route can be set faster than the devices and the control system can update the necessary parts of the state space. This will not affect the safety of simulation, but can affect the liveness.

If a tram event is 1 the sensor thread can interpret the STS, but if the interpretation has no effect on the state space because the conditions of the involved STSs are not evaluated to true a problem arise. In this case the tram event variable will be set to zero possible causing a deadlock as the "if then else" part of that sensor thread can no longer be entered - meaning that the sensor will not become high and the counter will not be increased.

The exception to the setting of the tram events is for sensors with a signal. A while loop in each route thread prevents the tram driver from crossing a sensor with a red signal[9].

[8] Mutex is short for mutual exclusion and the lock is a feature of the POSIX threads library.
[9] Such a loop is called a spin loop.

In the "real world" each device would have its own processor. Thus, the timing of the devices in the simulation are not as fast as it would be in the "real world". Calculations regarding how fast the tram events can happen in the "real world" yields a lower bound saying that it takes at least 1 second from the tram's front end passes a sensor until it can pass the next sensor[10]. Unfortunately, the simulation program is not fast enough for this. Inspection has showed that it takes the simulation 3 seconds in the worst case to update all necessary variables when a tram event has been set.

Nevertheless, the simulation is still useful as it shows the events of actual traversals of routes.

17.8 Chapter Summary

In this chapter a program simulating the traversal of routes in the running example network has been presented. The specific simulation of the concurrent interpretation of STSs, the simulation of tram events, and the simulation of route traversals have also been presented. Additionally, certain problem with the simulation has been presented.

[10] Maximum speed 70 km/h, minimal tram length 20 m, and minimal distance between sensors of 20 m.

Chapter 18

Discussion

In this chapter some of the aspects of the development will be discussed, detailing alternative choices and their pros and cons. In section 18.1 the modelling will be commented on, in section 18.2 the development method, in section 18.3 the verification process, and in section 18.4 the implementation.

18.1 Modelling Options

In this section the construction of the models is discussed.

18.1.1 The Modelling Method

The modelling of networks has been performed based on segments using sensors as connectors. Thus, only one type of building block exists, but their connectors[1] have different properties of how many building blocks they can connect to.

An alternative to this model was considered:

Brio Model. Referring to the well-known children's toy railway network, this model consists of several different building blocks, being one for each purpose in the network. Thus, a building block exists for the rails at the border of a network with one connector, one exists for a junction having three connectors, one exists for normal linear rails in the middle of a network with two connectors, and one exists for crossings with four connectors. The main advantages of this model of the network is that it is intuitively easy to grasp, and that the topology of the network can easily be described.

Unfortunately, the running example network (see J.2) does not easily facilitate such a model due to how points are connected in this specific network. This is a problem as the network is from the domain.

Several solutions to this problem have been considered:

- Adding extra sensors around non-controllable points to make the use of junction building blocks possible. This is the easiest solution to the problem but it has the

[1]By connector is meant the position where building blocks can be connected.

distinct disadvantage that some of the networks the domain specific language is intended for would not be in the domain.

- Adding an extra connector type. This connector type would then describe the network topology together with the sensors. Such connectors would be placed around non-controllable points. This solution would add unnecessary complexity to the model as an extra RSL type would have to be specified, with axioms and observers describing its properties and the relation to sensors.
- Further expanding the number of building blocks. This solution would lead to a large number of distinct building blocks. Actually, the number of distinct building blocks would correspond to the different roles of sensors in the chosen model. The problem with this model is the sheer size of the network description. The model size explosion which is present in the generator model of the chosen model would be present in the basic network description, and there is no reason to believe that the generator model would be any simpler.

None of these solutions were found to be usable, and therefore the model of the network has the topology described by the properties of the connectors and not the building blocks. Furthermore, the chosen model includes a property for each segment describing whether it crosses another segment or not.

This model is a little bit more abstract, and may not be as easy to grasp, but the network topology of the example network can be described in a satisfactory way.

An example illustration of points in the model based on linear segments and the brio-model is presented in figure 18.1.

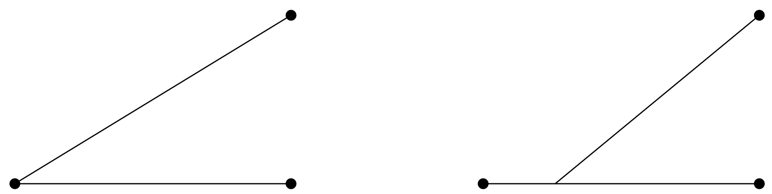

Figure 18.1: Segment based model of junction to the left and brio based model to the right.

An alternative model based on the BRIO approach is available at the following URL:

www.student.dtu.dk/~s973588/rslmodels/

The model is not verified in any way, and is not complete.

18.1.2 Connection of Points

The chosen model facilitates direct connection of points, controllable or non-controllable thus a sensor can be a point sensor for more than one point. Unfortunately it has resulted in a large number of different connector roles for sensors, which has made the model more complicated. By demanding that two sensors cannot be point sensors for the same point this would have been avoided. The reason for allowing direct connection of points is that this way it is possible to describe the network of the running example, which is a concrete network of the considered domain, without introducing additional sensors.

18.1.3 Direction in the Network

It has been chosen not to make segments directed, as the segments of the domain are not physically directed making it impossible to enforce a direction in the uncontrolled network.

Nevertheless, by introducing a logical direction through controllable and non-controllable points for routes, a global direction in controlled networks containing one or more points is introduced. All the considered networks have points, as otherwise they would be uncontrolled linear pieces of rail. Therefore, this notion of direction is sufficient in the model. Furthermore, the logical purpose of points supports this demand on routes.

18.1.4 Signals

The position of a signal is restricted to be at sensors with a neighbouring entry border sensor. This specific position has been chosen to model the fact that signals should only be placed at the start of a route. Furthermore, it is demanded that all routes have an entry signal, thereby, it is stated uniquely where signals are located in the network. Thus, the person planning routes in a network cannot directly place signals, as they will be positioned according to the routes.

For the direction of signals it has been decided that they are visible from the border sensor. Furthermore, axioms on the number of segments in the network ensure that this definition is unique.

Another possibility is to directly model the signal direction using an observer:

```
value
  signal_direction : SignalId → SensorId-set
```

This observer would then yield the sensors following the signal, being the sensors that are reachable if the signal shows *go*. This way of viewing the problem originates from the 2D illustration of tramway networks where signals actually point in a specific direction. The problem with this observer is that, because the signal can be placed at the stem sensor of point, more than one sensor may become reachable when the signals show *go*, hence, the SensorId-set of the function signature.

Yet another possibility is using another observer:

```
value
  signal_visibility : SignalId → SensorId
```

This observer would yield the border sensor from where the signal aspect is visible. This corresponds to the chosen method, but has the advantage of being able to uniquely model the visibility of signals in networks with less than three segments. On the other hand networks with less than three segments are not interesting in the scope of a control system, and therefore the definition of signal visibility is defined implicitly by the position of the signal.

18.1.5 The Global State & STS-L

The state space is distributed and concrete from the first RSL specification. STSs has a well-defined syntax and semantics and the control components and physical environment have been modelled by state transition diagrams (see chapter 6) before actually specifying the state space and the STS-L. Thus, the variables needed and their possible values had already been identified in the modelling of the different STSs, making a development from an initial abstract state space to a concrete distributed state space less important.

As mentioned above the STSs have a well-defined syntax and semantics - making the specification a matter of capturing this in RSL and identifying the details needed in the specific modelled STSs. Thus, an initial abstract model has not been created for the STS-L.

18.1.6 Alternative Method for Language Design

The modelling of the DST-L is performed using an alternative approach to language construction. This was described in section 5.4.

Several methods have been considered in the modelling of the DST-L.

It has been considered whether to make the types of the devices minimal, meaning only spanning over identifiers for devices in the considered network, or whether they should be general types. It has also been considered whether a type Network and Tables should be introduced or not. The chosen model design has minimal types and no Network or Tables type, thereby, differing from traditional language design in RSL.

- If the types where not minimal it would be necessary with additional pre conditions making functions partial.
- If a Network and Tables type where introduced it would be necessary with additional parameters for the functions[2].

In the final model there is a scheme representing the network and a scheme representing the tables. No actual types Network or Tables are in the model. The schemes only include identifiers for the devices being part of the considered network to avoid partial observers. Therefore, the network and tables is represented by functions mapping each identifier to the data for the device.

18.2 Development Method

It has been chosen to perform the specification of all parts of the model in the formal specification language *RSL*. This choice is based on the fact that the modelled system has safety critical aspect and the formal approach is therefore to prefer in favor of for example *UML*, as the semantics of the model cannot be precisely defined in *UML*. Other formal specification languages are available such as *VDM* and *CASL*, but *RSL* has the distinct advantage that we already possessed skills in constructing models in *RSL* - making it easier to focus on the quality of the model, instead of using resources on how to actually constructing it in the language.

Additionally, a formal model is the starting point of a formal verification of the model. The tool *eden* even has facilities for helping in the verification of *RSL* models, and therefore *RSL* was chosen.

The *RAISE* method provides a well-defined way of constructing models for implementation through step-wise development. The method has been used as a guideline in the

[2]Note that with Network and Tables types it would not make sense to use minimal types for devices

development without adhering completely to its principles. For instance the STS-L has not been modelled abstractly, thereby, skipping the first step of the development according to the method. The reasoning for this is presented in section 18.1.5.

18.3 Verification

In this section important aspects of the verification of the safety of the modelled system are discussed.

18.3.1 The Verification Approach

Safety has not been formally verified. A formal verification would be an obvious future task. An informal argumentation pro the model fulfilling certain safety requirements based on an inductive procedure with a strengthening of the invariant by the predicate consistent (see chapter 10) has been performed.

The approach of a verification process can be decided by pairing observers with the generators possibly having influence on the result of the observer. In the model observers and generators of the different objects exist, but since the guard functionality is represented by the conditions in the rules of the STS components, it is necessary to perform the verification at STS rule level. The different proof obligations consider the STSs instead and the use of the interpret_sts function.

Due to this effect of the model of the STS-L, conducting a complete formal verification would incur a considerable workload. This is because a decomposition of the proofs is difficult and because at least one generic control component would have to be unfolded in proving each second and third strong axiom.

18.3.2 Model for Verification

For a verification to be more manageable the STS-L or the *generator* would have to be modified or the control system would have to be modelled in some other way.

Modifying the STS-L is not a viable solution as the syntax and semantics of STSs are well-defined and the STS-L model closely follows these. Thus, an attempt to change the STS-L to accommodate a more manageable verification is not likely to be successful.

The *generator* could be modified by introducing a generator function for each different type of rule. Thus, a lot of generators and consequently also a lot of second and third strong axioms would be introduced. Hence, the proof obligations would be more decomposed - making it easier to conduct formal verification for at least some of the axioms. In retrospective this way of modelling the generator is probably a better choice than the chosen model - at least if verification is the main purpose.

Generally, it is a trade off between a model only intended for verification or a model also intended for implementation. In this thesis the main purpose has been implementation which is mirrored in the models.

18.4 Implementation

In this section important choices concerning and aspects of the implementation is presented and discussed.

18.4.1 The DST-L

The implementation of the DST-L is performed using the extensible markup language (XML). The language provides easy access to a structured way of representing data syntax in DTD's or XML schemas. Furthermore, XSLT provides a way of implementing observers and generators through XPath expressions. Thus, the necessary tools are available

The DST-L could have been implemented another way, possibly using a basic text-document and defining the syntax through some high level programming language where observers and generators could also have been implemented.

Another possibility could be using some kind of relational database system. Observers and axioms would then be implemented as SQL queries to the database. The obvious disadvantage is that a database is not portable like an XML document, but would have to be installed on the hardware.

The XML approach was chosen because of its accessibility, its widespread usage, and the fact that a simple web-browser, probably the most common computer application, can be used to view and modify the data[3].

Another advantage of implementing RSL specification in XML is that a refinement step to an imperative RSL model is not mandatory. This is because in XML it is possible to define unique identifiers for all types of objects and to use these identifiers to reference objects and this way ensure that for example a sensor defining a route actually is a sensor in the network. This corresponds to the use of the identifiers in the applicative RSL model.

18.4.2 The Generator

Since the DST-L was implemented using XML, it was obvious to use XSLT to implement the Generator, since the force of XSLT is to transform data in an XML document into new XML documents or other formats.

18.4.3 The STS-L

The choice of programming language for implementing the STS-L is based on two aspects.

As mentioned in section 1.1 the thesis is an extension of an ongoing research collaboration between the thesis supervisor and Jan Peleska. Hence, it is the intention that the product of the thesis being models, safety argumentation, implementation, and the description of the problem can be used in the continued research. In [Pel01] Jan Peleska presents a prototype implementation of the control system using state machines instead of STSs. This prototype was implemented with the thought of conducting some kind of model checking. Even though this was before the usage of STSs was introduced, the implementation in this thesis

[3]Since XForms is still a new technology it is not yet integrated in web-browsers, but it should only a matter of time.

uses the same programming language, a similar program structure, and design of functions with the hope that the intended model checking is also feasible with this implementation without too many alterations.

Additionally, the control system communicates with hardware devices. To make the communication easy the implementation should be as close to the hardware as possible. Based on this some assembler language would be a reasonable choice, but due to lack of experience in using such languages, C was chosen because it is one of the most basic of the high-level programming languages. Furthermore, C facilitates bits programming which is used extensively in the implementation.

18.5 The Concept

The concept is to generate control systems for specific networks based on a description in a DST-L. Thus, with this method a specific control system can be generated very fast. Normally, the process of constructing a control system for a network would involve both analysis, design, and implementation. This has all been performed generically in this thesis, thereby, eliminating the need for doing it for each network. Obviously, the construction of a generic control system is a lot more time consuming than for a single network, but by doing the work once and for all, it quickly becomes possible to save time. Constructing only a few control systems with this method may not justify the effort, but if the method becomes more widespread the time saved is substantial.

Chapter 19

Extensions and Future Work

In this section extensions to the model, the verification, and the implementation are presented and commented on.

The first six sections 19.1 - 19.6 considers how to extend the RSL models such that they include the simplifications presented in section 3.5. In section 19.7 - 19.11 additional possibilities of extensions to the models, verification, implementation, testing, and simulation are presented.

19.1 Additional Signal Aspects

In the model only two signal aspects are considered, namely *go* and *stop*. In the informal description additional signal aspects are mentioned, *go_straight*, *go_left*, and *go_right*. An obvious extension to the model would be introducing these additional signal settings. Introducing them has little influence on the control system and the safety of the network, but since they are present in the domain it would be a more correct model. Introducing them in the model would amount to introducing additional signal settings in the schemes TABLES1 and TABLES2 this way allowing them to be represented in the signal setting table.

```
type
  SignalSetting == GO_STRAIGHT | GO_LEFT | GO_RIGHT | STOP
```

The main problem with this procedure is that the network topology does not discern between left and right sensor of a junction, referring to both as turn sensor. This approach was chosen to make the description of the network as abstract as possible and the information of whether the turn sensor is actually on the right or left side of the straight sensor is not in any way important for the system. Therefore, a direction would have to be introduced to be able to discern between left and right.

A reasonable extension according to the presented model would be:

```
type
  SignalSetting == GO_STRAIGHT | GO_TURN | STOP
```

This corresponds well to the intention of the added signals and can easily be added to the models. Doing so would cause a modification of the observer signal_setting_is_go, the addition of an axiom, and the corresponding signal states would have to be added to the abstract syntax of the STS-L.

19.2 Onboard Computer

The models only consider trams with onboard computers for communication with the control system. The main purpose of the onboard computer is the request of routes. Other uses could be internal indication of signals[1] or alarm signals in each tram - making it possible for a safety monitor to signal directly to the tram driver if the network became unsafe.

To model both trams with and without computer access several issues would have to be taken into consideration. First of all the shared control of points would demand some kind of access priority. Consider the situation where two trams are positioned in front of the route start. The first tram has no computer, but the second does. The tram driver of the first tram leaves his tram to switch the points on the route, but simultaneously the second tram's driver requests the route via the control system. Which tram will get route allocation? This issue is closely connected with the simplification *extended driver protocol*, because if the drivers adhere to this protocol the issue will only be of sharing control. Thus, in that situation it is not necessary to decide the prioriy, but only how to share the control.

Inspired by theory of parallel systems concerning mutual exclussion [And00] sharing control could be modelled by an observer locked on controllable points:

type
  LockId == TRAM_DRIVER | SYSTEM

value
  locked : PointId → **Bool** × LockId

Thus, there are two ways for the route dispatcher to reserve a route.

1. The route is requested.
2. The points on the route is set and locked by the tram driver.

In the first case the route dispatcher will lock the points, and the route controller will perform the switching and at time of deallocation unlock the points. In the second case the route dispatcher will reserve the route and the point setting part of the route controller will be skipped, meaning that the tram driver will have to unlock the points as he passes them.

Obviously, this way of running the tramway network is not usable if the route is long and if many points exists on the route the tram driver will have to walk quite a distance. On the other hand the running example network only has one point on each route located right at the start of the route, meaning in that network the method is usable.

Such a model would also lead to a modified state space, where the information about the lock is added to each point:

value
  PointStates = {| m : STSLPointId $\overrightarrow{m}$ (PointState × PointState × **Bool** × LockId)
      • **dom** m = { x | x : STSLPointId} | }

This concept of sharing control of points between system and driver has not been fully developed, but it is not the intention of the system, and as can be deducted from the description above it will complicate the control components.

[1]Could be usable in situation where vision is impaired such a foggy weather

19.3 Only One Tram on a Route

In the model only one tram is allowed on a route at a time. To allow multiple trams on a route, it would be necessary to make some corrections to the control components.

- The counter controllers need no modifications to facilitate multiple trams on a route.
- The route dispatcher ensures that no conflicting routes are reserved. Routes does not conflict with themselves, therefore the route dispatcher does not need to be modified.
- If a route being requested is already in use, the route controller of the route is in state *allocated_in_ use*. Since the route is in use, it is allocated. The allocation has also been closed by setting the signal aspect of the route to *stop*. It is therefore necessary to reopen the allocation to allow multiple trams entering the route.

In figure 19.1 a suggestion on how the route dispatcher for route R1 of the running example must be modified to facilitate multiple trams on routes.

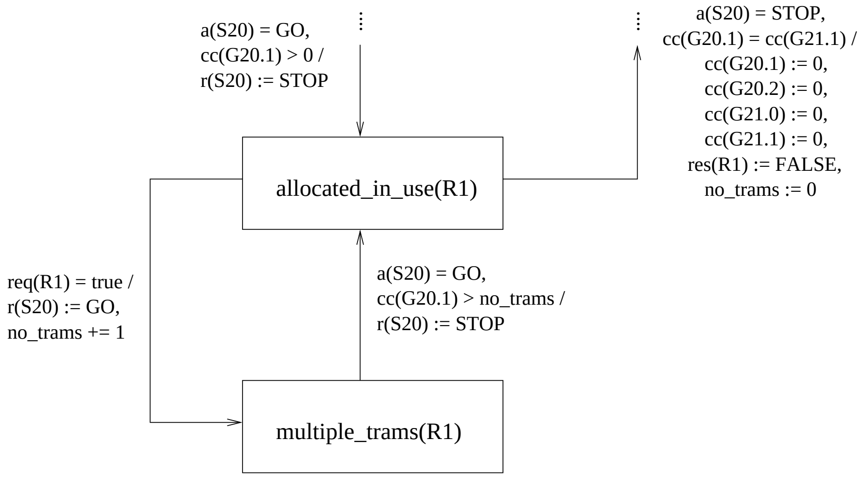

Figure 19.1: Modification of route controller to facilitate multiple trams on routes.

The variable *no_ trams* is the number of requests of a route that has happened while it is allocated and in use. Consequently, the variable is set to zero when the route is deallocated

Several issues must be considered before allowing multiple trams on routes. For example often used routes could block the rest of the network. These issues are not considered here.

19.4 Extended Driver Protocol

The problem avoided by the *extended driver protocol*, is that trams do not drive the route they request. Thus, it is not a safety problem. Depending on the amount of traffic in a specific network it may never occur that two trams request a route starting at the same sensor simultaneously, and it may never occur that a tram request a route it is not ready to drive.

The system could be modified to ensure correct reservation without the *extended driver protocol*. Two options are considered:

- With onboard computers the tram request could be made automatic such that it must happen before the tram has fully passed the entry border sensor. This would lead to an additional assumption about the timing of sensors. This way the control system could use a list of pending requests to ensure valid evaluation.

- Without computer requests of routes, the tram would need some kind of tram identification. The control system would then have to keep a list of which trams are on the segments and their order to ensure the right order of treating requests.

Either way the system would have knowledge of which trams are in front of the different entry signals and in which order.

19.5 Introducing Time

To simplify the model it was chosen not to model time. Time would add en extra dimension to the model and allow more explicit simulation of devices, including the concept of failing devices.

Not modelling the issue of time has made it necessary informally to argue pro that a number of timing conditions are reasonable.

The extension to RSL, called Timed RSL (TRSL) [CWG04], has a built-in way of modelling time. The idea is to develop the models using the normal RAISE method, reaching an imperative concurrent specification. Thus, TRSL could have been used if concurrency of the different devices had been modelled. But, since the main product is not the simulation this has not been done. This is clearly possible future work, modelling the simulation this way allowing the timing assumptions to be modelled using wait statements of TRSL.

19.6 Safety Monitor

The safety monitor control component has been left out of the model to simplify it. An extension would be modelling and implementing the safety monitor.

The safety monitor has two purposes:

Check for failing devices. The safety monitor checks if the switching of a device once requested is completed within a given time limit. If not the safety monitor deems the network unsafe and sets all devices according to the stable safe state[2]. To model this, time would have to be introduced in the model (see section 19.5).

Cyclic checks for safety. In [HP02] this functionality of the safety monitor is described. The intention is that if trams at a give time drive on conflicting routes the safety monitor will notice this and put the system in stable safe state.

Unfortunately, the approach presented in [HP02] is too restrictive as two trams driving non-conflicting routes in some cases can make the safety monitor react. The suggested safety monitor uses the route entry sensor to determine whether two conflicting routes is driven at the same time making the network unsafe. Thus, according to the article the running example network is unsafe if the control counters for sensor *G20.1* and *G22.1* are both larger than zero. It is correct that this could be the case of route 2 and route 3 being

[2]This state corresponds to the initial state of the system.

driven simultaneously which is clearly unsafe as these routes are conflicting. But on the other hand it could also be the case of route 1 and route 3 being driven simultaneously, which should be allowed.

Two possible solutions to this problem are proposed:

1. Using the control counter of the first sensor in the route sequence, that is not part of other routes, instead of the start sensor would solve the problem.
2. Another possibility is using the control counter of the start sensor and the position of points to derive the safety conditions.

A safety monitor STS for the running example using the first solution proposal can be presented by the state transition diagram in figure 19.2, where *reset variables* denotes the setting of all variables to their initial state.

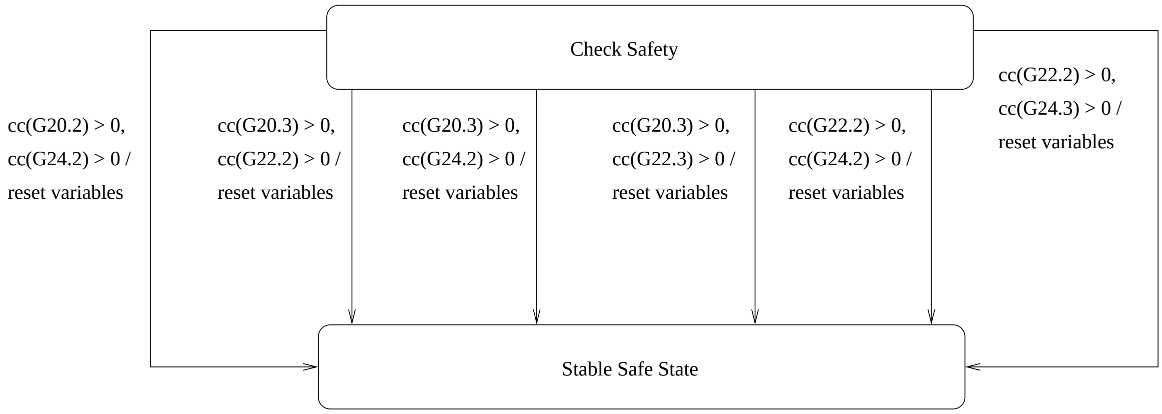

Figure 19.2: Safety monitor for running example network.

Using the first solution proposal it means that conflicts are detected later than if using the second proposal. This is the case if the distance from the sensor on which the test is performed to the start sensor of the route is long.

It is also possible that a route is defined by a sequence of sensors which all are part of another route. Then it is not possible to express the first solution proposal.

19.7 Test

The test of the system was based on a structural and a functional test. The structural test was not complete, which certainly should be the case in connection with a "real world" implementation. In our case a simulator but not a "real world" system has been implemented, and therefore it should be reasonable to concentrate on the functional test.

19.8 Simulation

The simulation of the working tramway network has already been discussed to some extend in chapter 17. The main focus of this section is the simulation of devices by STSs and a construction of a GUI for the simulation.

19.8.1 Failing Devices

The STSs simulating devices do not consider the failure of the device. As mentioned in 19.6 the safety monitor can be generated with a means of modelling failing devices if time was modelled. In that situation a failing device is a device that has not updated its state upon request from the control system within a given time limit.

Another possibility is introducing an extra control mode variable state for STS devices named UNDEFINED. The transition to this state could then for instance be based on probability or it could be controlled in the simulation. If the safety monitor then checked for UNDEFINED devices, it could be simulated how the control system would handle the situation of failing devices.

19.8.2 GUI for Simulation

The simulation of the control system is an event based print out. The simulation is not interactive as the events of trams passing a sensor is written in a script in a separate thread.

Future work could include a graphical user interface where events could be viewed in a 2D representation of the network through moving trams and different color codes deciding the state of devices. Possibly, a graphical representation of the STSs using state transition diagram could also be presented for the user. Furthermore, the simulation could be made interactive letting the user manipulate tram events either by moving a tram or by setting the variables in a command prompt.

19.9 Advanced Control Algorithm

The modelled control algorithm, named the interpreter, iteratively runs through the STSs and their rules looking for one that can fire. Thus, a lot of conditions is examined that will not be fulfilled. Another approach could be only interpreting the route controller STSs for reserved routes and possibly also only interpreting counter controller STSs for sensors in reserved routes[3]. This way the algorithm is opimitised and the performance is thereby improved corresponding to the size of the considered network.

Such an optimisation of an algorithm is future work, and other optimisations of the system may be discovered.

19.10 Formal Proof of the Model

If the control system was to be used in a ”real world” setting and not a simulation, a complete formal verification would be desirable. As argued in section 18.3 the model could be optimised for verification, but doing so could impair the implementation. Nevertheless, the verification is important future work, if the system is to be used in practise.

[3]Somewhat dangerous, if the system should fail and not in accordance with the safety monitor.

19.11 Real World System

The ”real world” physical environment consists of actual physical entities. The control system would communicate with the devices through masks translating between the system's and the hardware's language. Thus, an obvious future task is constructing the specific masks for identified hardware and testing the control system in an actual network. Such a test is clearly beyond the scope of this thesis, but is mandatory before using the software in practise.

Chapter 20

Conclusion

In this chapter the main results of this thesis are summarised.

20.1 Results

The results of the thesis are presented in this section. In section 20.1.1 the results of the modelling is presented, in section 20.1.2 the results of the verification is presented, and in section 20.1.3 the results of the implementation is presented.

20.1.1 Modelling

The formal model of the DST-L is developed in two steps. Initially, an abstract specification of the network part of the DST-L is used to identify properties of the network, and the specification is concretized according to the identified properties. The development is similar for the tables part of the DST-L, with the difference that the properties are well-known prior to the construction of the abstract model. Thus, the abstract model of the tables part is used to make certain that all necessary properties are identified and to ensure that the development of the two DST-L parts is similar.

The generic control components of the control system are represented by STSs. This is a good solution making it possible effectively to control the interpretation of the control components according to a well-defined semantics. Additionally, to simulate the running system the physical environment is also represented by STSs.

The model of the STS-L is specified as a minimal model, thereby, only facilitating the construction of the necessary STSs to construct the control components and the physical environment. Consequently, the model is more precise and concise in the context of the tramway domain, than a more generic STS-L model.

The modelling of the STS-L follows the traditional language approach in RSL. The syntax and the semantics of the language are well-known, and therefore the initial model is concrete.

The model of the different generic STS control components and the STSs simulating the physical environment is parameterised with the DST-L model, thereby, it becomes a generator of control systems for specific networks. To gain an overview of the different STSs each STS is modelled by an RSL value.

20.1.2 Verification

Predicates expressing safety are formulated and proof obligations for the safety of the tramway network controlled by the generic control system are constructed. Thus, the foundation for a formal verification is presented.

Furthermore, an informal argumentation that the proof obligations are fullfilled is presented.

20.1.3 Implementation

In general the implementation follows the models and thereby the transition between model and implementation is clear.

The implementation of the DST-L is performed using XML and XSLT. This is a good choice, as XML is a powerful tool for representing structured data. The syntax of the language is easily formulated by declaring a DTD and the static semantics is expressed by an XSL transformation testing the data in the description stored in an XML document.

The models of the generator and the operational semantics of the STS-L are developed to imperative models.

The description of specific tramway networks in the DST-L is given by an XML document. Therefore, it is obvious to implement the generator using XSLT which facilitates the transformation of XML documents to other formats.

The control system represents a prototype of a ”real world” control system. Therefore it is implemented in the imperative programming language C, facilitating the uses of bit programming in the implementation.

20.2 The Concept

The fundamental concept is developing domain specific computer systems using a domain specific language. The concrete domain is tramway networks and the system a control system.

A generic control system for tramway networks is presented in this thesis. From the description of a specific tramway network in the domain specific language, configuration data for the generic control system is generated. Based on this data a control system for the specific tramway network can be initialized.

The concept must be evaluated as being a success. A lot of time is saved when constructing control systems for more than a few tramway networks. Thus, if multiple control systems are generated using this method, the resources put into constructing the generic system are saved.

Index

Bibliography

[And00] G. R. Andrews. *Foundation of Multithreaded, Parallel, and Distributed Programming.* Addison-Wesley, 2000.

[CWG04] A. E. Haxthausen C. W. George. UML-ising Formal Techniques. In *3rd International Workshop on Integration of Specification Techniques for Applications in Engineering (INT-Workshop), Barcelona, Spain*, March 28 2004. only chapter 6.

[Dub03] M. Dubinko. *XForms - Essentials.* O'Reilly, 2003.

[Eng03] Morten Engelbrecht. Moderne sporvogne i københavn - en gevinst på alle måder. 2003.

[GHH+95] C. George, A. E. Haxthausen, S. Huges, R. Milne, S. Prehn, and J. S. Pedersen. *The RAISE Development Method.* Prentice Hall, 1995.

[Gja02] T. Gjaldbæk. Modelling railway interlocking systems. Master's thesis, DTU, 2002.

[GJM03] C Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering.* Prentice Hall, 2003.

[Gro92] The RAISE Language Group. *The RAISE Specification Language.* Prentice Hall, 1992.

[Hax03] A. E. Haxthausen. Idpt state machines. IMM, 2003.

[HM02] Elliot Rusty Harrold and W. Scott Means. *XML in a nutshell - a Desktop Quick Reference.* O'REILLY, 2002.

[HP00] A. E. Haxthausen and J. Peleska. Formal Development and Verification of a Distributed Railway Control System. *IEEE Transaction on Software Engineering*, 26(8):687–701, 2000.

[HP02] A. E. Haxthausen and J. Peleska. A Domain Specific Language for Railway Control Systems. In *Proceedings of the Sixth Biennial World Conference on Integrated Design and Process Technology, (IDPT2002), Pasadena, California*, June 23-28 2002.

[HP03a] A. E. Haxthausen and J. Peleska. Automatic Verification, Validation and Test for Railway Control Systems based on Domain-Specific Descriptions. In *Proceedings of the 10th IFAC Symposium on Control in Transportation Systems.* Elsevier Science Ltd, Oxford, 2003. To appear (preprints exist), ISBN 0-08-044059-2.

[HP03b] A. E. Haxthausen and J. Peleska. Generation of Executable Railway Control Components from Domain-Specific Descriptions. In *Proceedings of the Symposium on Formal Methods for Railway Operation and Control Systems (FORMS'2003)*, pages 83–90. L'Harmattan Hongrie, 2003.

[KR88] B. W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice Hall, 1988.

[LV00] M. P. Lindegaard and P. Viuf. Modelling railway interlocking systems. Master's thesis, DTU, 2000.

[Pel01] J Peleska. *Projekt EDASS Softwareentwurf (German)*, 2001. Documentation of C-implementation of Control System.

[W3Ca] http://www.w3.org/tr/xforms.

[W3Cb] http://www.w3.org/xml/.

[W3Cc] http://www.w3.org/style/xsl.