

A Thesis Submitted for the Degree of PhD at the University of Warwick

Permanent WRAP URL:

<http://wrap.warwick.ac.uk/110541/>

Copyright and reuse:

This thesis is made available online and is protected by original copyright.

Please scroll down to view the document itself.

Please refer to the repository record for this item for information to help you to cite it.

Our policy information is available from the repository home page.

For more information, please contact the WRAP Team at: wrap@warwick.ac.uk

**A Formal Theory of Railway Track Networks
in Higher-order Logic
and its Applications in Interlocking Design**

Wai Wong B.Sc. M.Sc.

A dissertation submitted for the degree of
Doctor of Philosophy in Engineering
to
the University of Warwick

Department of Engineering,
University of Warwick.
Coventry, England

February 1992



to

Shu Fong and Anton

Contents

Acknowledgement	x
Declaration	xi
Summary	xii
I Preliminary	1
1 Introduction	3
1.1 Safety-critical systems	4
1.2 Formal methods	5
1.3 An overview of the research	10
2 The HOL Logic and the HOL system	12
2.1 ML — The meta-language	13
2.2 Overview of the HOL Logic	17
2.2.1 HOL types	18
2.2.2 HOL Terms	20
2.2.3 Theories	22
2.2.4 Theorems	26

2.3 Theorem proving in HOL	28
2.3.1 Forward proof	28
2.3.2 Goal directed proof	29
3 Principles of Railway Signalling	31
3.1 Railway track and signalling equipment	32
3.1.1 Track components	32
3.1.2 Signals	35
3.2 Signalling principles	37
3.3 Solid-state interlocking	39
3.3.1 Features of the SSI	40
3.3.2 The design procedures	42
3.3.3 The benefit and the future	43
II Theories	45
4 The mathematical foundation	47
4.1 The theory graph	48
4.1.1 The representation of graphs	48
4.1.2 Some basic definitions of graph	51
4.1.3 Relationship between vertices and edges	52
4.1.4 Operations on graphs	55
4.1.5 Subgraphs and graph isomorphism	59
4.2 The theory path	63
4.2.1 Walks in a graph	63

4.2.2	Some operations and facts on sequences of edges	64
4.2.3	Trails and paths	68
4.2.4	Some properties of paths	69
5	Modelling of Railway Components	73
5.1	The theory TRACK	74
5.1.1	Joins	74
5.1.2	Track circuits	75
5.1.3	Points	76
5.2	The theory SIGNAL	77
5.2.1	Simple signals	78
5.2.2	Compound signals	80
5.3	The theory PART	82
5.3.1	Parts	82
5.3.2	Edge labels	83
6	The network model	86
6.1	Specification of railway track networks	87
6.2	Examples of networks	90
6.3	Inductive reasoning on networks	95
6.4	Some properties of networks	98
III	Applications	105
7	Verification of track layout	106

7.1	Layout compiler	108
7.2	The network verifier	109
7.3	Formal specification of track layout	112
7.3.1	Syntax	112
7.3.2	Semantics	112
7.4	The implementation of the verifier	117
7.4.1	The parser	117
7.4.2	The prover	119
8	Generation of control tables	125
8.1	Definition of routes	126
8.2	Finding routes	129
8.3	Automatic control table generation	131
9	Interlockings and state machines	137
9.1	States of a network	137
9.2	Proving routes	144
9.3	Interlockings	145
9.3.1	State machine theories	146
9.3.2	Interlocking as state machine	148
10	Conclusions and future research	153
10.1	A generic abstract model	153
10.2	Applications of the model	154
10.3	The HQL system	157

10.4 Methodology issues	159
A HOL theories	189
A.1 The theory func	169
A.2 The theory graph	171
A.3 The theory elist	179
A.4 The theory path	181
A.5 The theory SIGNAL	185
A.6 The theory TRACK	194
A.7 The theory PART	199
A.8 The theory NETWORK	203
B ML source listings	205
B.1 The file mk_func.ml	205
B.2 The file mk_graph.ml	209
B.3 The file mk_subgraph.ml	222
B.4 The file mk_elist.ml	227
B.5 The file mk_path.ml	232
B.6 The file mk_signal.ml	241
B.7 The file mk_track.ml	247
B.8 The file mk_part.ml	250
B.9 The file mk_network.ml	252
C Listings of the verifier	262
C.1 The file rail.grm	262

C.2 The file <code>rail_help.ml</code>	263
C.3 The file <code>rail_load.ml</code>	267
C.4 The file <code>var_network.ml</code>	268
C.5 The file <code>mk_verifier.ml</code>	274
C.6 The file <code>Makefile</code>	275
 D Routes and control tables	 277
 E Level crossing—a case study	 281
 F Dynamic networks and state machines	 284
F.1 The file <code>dnetwork.ml</code>	284
F.2 The file <code>setlist.ml</code>	285
 Index	 288

List of Figures

1.1	System specification and verification.	7
3.1	A slip crossing and its equivalent atomic parts.	32
3.2	The principle of track circuit.	34
3.3	An overlap.	35
3.4	An example of track layout: a double left-hand junction.	37
II	The hierarchy of theories.	46
4.1	Examples of simple graph.	50
6.1	A simple network.	90
6.2	Another simple network.	91
6.3	A network formed using NJOIN.	92
6.4	A track layout containing a passing loop.	92
6.5	A network model representing the track layout in Figure 3.4.	94
6.6	Location of vertices: case 2.	102
7.1	Generation and verification of track layout.	107
7.2	Isomorphism between engineering design and theorem proving.	110
7.3	Syntax of Railway Layout Specification language.	113

7.4	Listing of <code>prove_network_njoin</code>	121
7.5	A HOL session of verifying network specification.	124
9.1	A graph representing the passing loop at <i>l</i>	144
9.2	An interlocking state machine.	148
9.3	A state transition diagram of generic interlocking machine.	150

List of Tables

2.1	HOL primitive terms.	20
2.2	HOL infix terms and special constants.	21
2.3	HOL binders.	21
3.1	Track components.	33
3.2	An example of control table.	39
5.1	Projection operators and predicates for :Part.	84
9.1	Abbreviated types for states and state functions.	139

Acknowledgement

I sincerely thank Professor W. J. Cullyer for his guidance and invaluable help and advice throughout the entire period of my study. He introduced me to the fields of formal methods and safety-critical systems, encouraged me to further the research. Without his help and encouragement, I would never be able to complete the study.

I would like to express my thanks to Dr. M. Gordon of University of Cambridge Computer Laboratory who kindly provided me a two-week study visit to the laboratory where I learnt considerable about the HOL system and to Dr. T. Melham of the same laboratory who gave me many valuable suggestions and helped me to solve some difficult technical problems in the development of the formal theories.

I am also in debt to Dr. A. Cribbens and Mr. I. Mitchell of British Rail Research who gave many suggestions and comments on the direction of the research and technical points about signalling systems.

Dr. J. F. Crain and Dr. N. Storey reviewed a draft of this dissertation and suggested many improvements.

Declaration

The author declares that the research described in this dissertation has been carried out by him except where explicitly stated otherwise.

The theories described in Part II have been presented to the *1991 International Workshop on HOL Theorem Proving System and its Application* and a paper published in the workshop's proceedings[68].

An outline of the research also appears in a joint paper with Prof. W. J. Cullyer submitted to the *IEE Computing & Control Engineering Journal*.

Summary

The research described in this dissertation centres on the application of a discipline of formal methods in railway signalling system design. A generic abstract model of railway track networks and signals has been developed in Higher-Order Logic(HOL). It consists of several theories arranged in a hierarchy. Railway track networks are modelled by a class of constraint labelled directed graphs. HOL theories of graphs and paths have been developed for representing track networks. HOL theories modelling individual track components and signals have also been developed. These theories are then combined to create a theory of track network.

Three applications of this model are described. The first is a network verifier which verifies a formal specification of track layout against its abstract model by proving theorems automatically. The second application is to extract information from the specifications and to create control tables automatically. Lastly, a method of modelling the interlocking processor using finite state machines is described.

Although this research has centred on railway signalling, it can be viewed as a case study of how to apply formal methods in the analysis and design of safety-critical systems. The approach and methods used can be generalized in order to be useful in other industries.

Part I

Preliminary

The main text of this thesis, which consists of ten chapters, is divided into three parts. The first part, **Preliminary**, contains introductory material. There are three chapters in this part. The first chapter begins with a brief discussion of safety-critical systems and formal methods and continues with an overview of the research. The major tool used in the research is Higher Order Logic (HOL). The second chapter provides a brief introduction to HOL logic and the HOL system for the benefit of readers who are not familiar with them. This chapter also explains the notation for presenting HOL text and examples used in subsequent chapters. Chapter 3 describes the basic principles of railway signalling and the state-of-the-art technology for the automation and integration of signalling systems.

The second part of the thesis, **Theories**, presents the HOL theories resulting from the research. These theories form a generic abstract model of railway track networks. The last part, **Applications** describes several possible applications of the theories in the specification and design of interlocking. The dissertation concludes

with a discussion of the findings of this research and suggestions for further works.

This dissertation also includes several appendices. These list the HOL theories, the ML sources of these theories and listings of various programs described in the main text.

Chapter 1

Introduction

This chapter gives a brief introduction to safety-critical systems and formal methods and an overview of the research presented in this dissertation.

During the last decade, advances in microelectronics and microcomputer technology have changed the way we work and live. Programmable devices have replaced much old, hard-wired equipment to offer improved flexibility and cost effectiveness. This provides the designer with many opportunities for developing new products and new manufacturing processes, and control systems containing programmable devices are now very common. Applications include the generation of electricity, flying large passenger aircraft and safeguarding the running of trains. As these systems become more powerful and perform more difficult tasks, they also become extremely complex. The complexities of some of these computer control systems have grown to a point where even their designers are barely able to comprehend them. This raises some serious questions: how can we be sure that the systems function correctly and what will happen if they fail?

The research to be described in this dissertation is a small contribution towards

an answer to these questions. Like designers in other well-established branches of engineering, computer control system engineers resort to mathematics. In this case, the helping hand comes from mathematical logic. Research in applying formal logic in the analysis and design of computer systems has been carried out for several decades, but it is only recently that the technology has matured to a point where it is feasible to use it in practical systems. Nevertheless, there are still many unsolved problems, especially when dealing with large complex systems. The research presented in this dissertation is a first attempt to apply a particular discipline of formal methods, namely Higher-Order Logic, to a specific problem—railway signalling systems. However, it is possible to generalize the methods used in this research so that they can be used in other types of system.

An overview of the research will be presented in the last section of this chapter, but before that, a brief discussion of safety-critical control systems and formal methods will be given.

1.1 Safety-critical systems

If, when a system fails, it causes human injury, or even fatality, or causes serious environmental damage, then such a system is a *safety-critical system*. Examples of such systems include shut-down systems for nuclear generation plants, flight control systems for passenger aircraft, railway signalling interlocking systems and radiotherapy equipment.

Safety-critical systems can be classified into different *risk classes* according to two orthogonal dimensions: the *severity* of potential accidents and the *frequency*

of their occurrence. The risk of a system is higher if the consequence is more severe or the frequency is higher. Preliminary hazard analysis should be carried out to assess the risk of a system before any extensive design and development work is commenced. There are international and national standards governing hazard analysis, classification of critical systems and, specification and design of safety-critical systems [41] [40] [52] [51] [37] [57]. Preventative measures should be taken in the design of such systems to minimize the risk.

After the risk associated with the failure of a system is identified, subsystems and components have to be classified according to their function criticality, i.e., the importance of each subsystem or component in ensuring the system safety. One method of categorising safety systems uses four classes: class I to class IV. The higher the class, the more critical it is. When analyzing railway signalling systems [24], Cullyer found that point and signal actuators are of class II, and the point position sensors and signal proving circuits are of class III. The sensors and proving circuits are more critical because, if they malfunction, the system loses the correct information of its state and hence, may perform a hazardous function without knowing it. For example, a broken RED bulb (actuator) can be detected by the proving circuit, but a faulty proving circuit may make the system think that the RED aspect is alright when, in fact, it is not.

1.2 Formal methods

Formal methods are the application of applied mathematics — formal logic — to the design and analysis of computer systems[58].

Every formal method is based on a formal system $\langle L, C \rangle$ where L is a language and C a consequence relation. A language is defined by a set of symbols S and a grammar which specifies the rules of forming sentences using symbols in S . A consequence relation is a set of inference rules which transform sentences in L while maintaining their validity. For example, *modus ponens* is an inference rule in classical predicate logic. A structure $\langle U, I \rangle$, where U is an universe containing a set of values and I is an interpretation mapping sentences of L into U , is used to assign meanings to sentences of L . A structure M is a model of a sentence A if A is true in M . A sentence A is valid if it is true in every model of L . This concise description of the theoretical bases of formal methods is due to Wing [19].

Formal methods are usually applied in three phases: formal specification; design and documentation; and verification. The requirements of a control system are usually first written in a natural language. In the formal specification phase, functional requirements are translated into a formal language. This is necessary because a formal language eliminates ambiguity. Implementation of the formal specification is developed in the design and documentation phase. In this phase, formal methods can be used to transform a specification into an implementation. This process is known as *synthesis*. The whole system is often divided into subsystems, and different technologies are used to implement various subsystems. For example, hardware can be used to implement a subsystem which requires a rapid response, while software is used for other part of the system. Different methods are often used with different implementation technologies. After the design has been developed, it has to be verified against the specification to see whether it really implements the specification correctly. This is the third phase. The verification is often carried out by proving

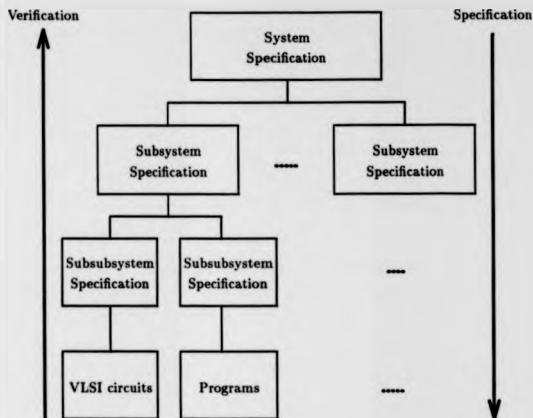


Figure 1.1: System specification and verification.

theorems asserting the equivalence of the specification and the implementation.

In developing large complex systems, the entire system is divided into subsystems and they in turn are divided into subsubsystems. The processes of specification, design and verification are often applied to several levels as illustrated in Figure 1.1. The implementation of a level is often the specification of the next lower level except the lowest one where physical VLSI circuits and actual program codes become the implementation. Verifications are carried out to show each level is correctly imple-

mented by its next lower level, e.g., a theorem asserting the functional equivalence of the system specification and the subsystem specifications.

The benefits of using formal methods are many, but the most important ones are:

- it helps designers to understand their problems more thoroughly, to gain greater insight, and thus to achieve better design;
- it helps to uncover more errors at earlier stages, and thus reduces development costs, and;
- it helps to give higher confidence in the correct functioning of the system.

All these help to reduce the risk of accident and save time and money, resulting in a better system.

Many formal methods have been developed during the last two decades. Now, some of them are becoming mature and are widely available and are being used in practical systems. Methods in this category include Z, VDM, OBJ, HOL, and the Boyer-Moore theorem prover. All these have supporting computer tools.

Z is a notation based on elementary set theory and first-order logic, and was developed at Oxford University's Programming Research Group in the late seventies and early eighties. A definitive description of Z can be found in [62], and examples of using Z can be found in [63] [27] and [28].

VDM stands for Vienna Development Method. It is a model-based specification language developed at the IBM Vienna Research Laboratories during the 1970s. In VDM the description of systems, both specification and designs, are given as models. The major references on VDM include [42] [6], [59] and [7].

OBJ is a specification language with both executable and non-executable parts based on order-sorted equational logic. It integrates specification, design, prototyping and verification in a single system. The algebraic approach to specification on which OBJ is based is described by Goguen *et al.* in [32]. The current version of OBJ is described in [30], and its use as a theorem prover in [31].

The Boyer-Moore theorem prover was developed by Boyer and Moore at the University of Texas at Austin and was started in 1972. It supports first-order logic using LISP as the meta-language. The logic and the theorem prover are described in [8] and [9], and the future of this ongoing project is discussed in a recent paper [10]. This prover has been used to verify a microprocessor design [39]. A collection of system components known as the CLI 'verified stack', in which a lower level component implements its next higher level component, has been verified and described in the *Journal of Automatic Reasoning* 5(4) by five related papers [5] [4] [38] [45] and [66].

HOL is a theorem prover supporting higher-order logic which has been used in the research to be described in this dissertation. A brief description of the logic and the system will be given in Chapter 2. Meanwhile, its application areas are reviewed briefly.

The first application area of HOL was initially in the specification and verification of hardware design. This was first advocated by Hanna [36]. The VIPER microprocessor was a processor whose functional requirements were formally specified [23] and this specification has been partially verified in HOL [15] [16]. Although the processor is very simple compared to most commercially available microprocessors, the methodology used for the specification and verification of the processor is

an excellent case study of the application of formal methods in real hardware, and this should be further developed and exploited.

Other areas of HOL applications include: software verification[33] and communication protocols[43]. The idea of producing a totally verified system by linking verified software to verified hardware has been explored by Joyce [44].

1.3 An overview of the research

Railway signalling systems are certainly safety-critical systems. The safety record of railway signalling systems has been very good due to the stringent requirements imposed on the design and manufacturing of such systems. From the early mechanical semaphore signals to modern power signals based on electromechanical technology, signalling engineers have had a great helping hand from nature, namely the gravitational force, in ensuring their systems are fail-safe. In semaphore signals, the weight of the arm forces it to return to the danger position in the event that the interlocking frame is broken. However, when computers and solid state components are introduced, this fail-safe feature, guaranteed by a natural force, is lost. Therefore, more rigorous methods of design and verification of signalling systems are necessary.

The research to be presented in this dissertation is thought to be the first attempt to apply formal methods to the design and verification of railway signalling systems.

The approach of the research is first to create a formal model of a railway track network and signals. This model is then used in the formal specification of railway track layouts and in other phases of interlocking system design.

After studying the basic principles of interlocking in signalling systems, the au-

thor developed a formal model to represent railway track networks and signals. The model consists of several HOL theories which form a hierarchy. The top level theory is the **NETWORK** theory which models the topological relation of the track network. Some generic properties of the model have been derived. Networks are represented by a class of constrained labelled directed graphs. A theory of labelled directed graphs has been developed for the purpose of representing track networks.

The first application of this formal model of track networks was to generate formal specifications of track layouts and to verify them. A verifier has been implemented to automate the process of verifying track layout specifications. The verification is performed by proving theorems stating that the given specification represents a well-formed track network.

Next, a method of generating control tables is developed. This method extracts information from the formal specification to compile the control tables. The specification of the core of a table generation program will be described.

The dynamic aspects of the track network and the problems of ensuring that the interlocking operates safely are investigated. The most important properties of a working network are *safety* and *liveness*. Safety is guaranteed by the correct implementation of the interlocking regulations. In essence, no conflicting movement of trains should be allowed. The liveness of a network can be expressed as the ability to run trains according to the prescribed timetable. A network with all signals constantly displaying the red aspect is certainly safe but not live. A method of modelling interlocking systems by finite state machine and how to deduce important properties of such machine will be described.

Chapter 2

The HOL Logic and the HOL system

This chapter gives a brief introduction to Higher-Order Logic and the HOL system. The notation for presenting HOL text in this dissertation is also described.

HOL stands for Higher Order Logic. The HOL logic is a version of typed predicate calculus based on the simple theory of types founded by the logician Alonzo Church[14]. A more modern description of simple type theory can be found in [3]. The HOL logic is supported by the HOL system¹ developed by a team headed by Gordon in the University of Cambridge Computer Laboratory[34]. For the sake of completeness, and for the benefit of readers who are not familiar with HOL, this chapter provides an informal description of both the logic and the HOL system. A formal, set theoretic based description of the HOL logic and the soundness of the

¹The acronym HOL is used for both the logic and the computer system supporting the logic. Usually, the context can make clear what is referred to when the acronym is used on its own. In cases where this is not clear, the term 'HOL logic' or 'HOL system' will be used instead.

HOL proof system can be found in [17], and tutorial examples of using HOL are given in [18].

The HOL system provides an interactive environment for the user to carry out formal reasoning in the HOL logic. The interface to the logic is the meta-language ML which is a functional programming language. The HOL system used in the research described by this thesis is HOL88 version 2.0.²

A brief introduction to the meta-language ML is given in the Section 2.1. The object language, the HOL logic, is described in the Section 2.2. Methods of carrying out formal proofs in HOL are described in the Section 2.3. At the same time as describing HOL, the notations for presenting HOL text and theorems in the subsequent chapters are introduced.

2.1 ML — The meta-language

ML is a strongly typed functional programming language. It was first designed and implemented by Milner, Morris and Wadsworth at the University of Edinburgh in the early 1970's. It was designed originally to be the meta-language of the Edinburgh LCF system[35]. Since then, the language has evolved to become a large family of related languages. The most notable one is the *Standard ML* defined by Milner *et al.*[49], and it has several very good implementations.³ The ML of the Cambridge

²In fact, the theories and proofs described in this dissertation were developed using Version 1.11. Small modifications were made to bring them up-to-date with Version 2.0 when it was released.

³Two new implementations of HOL based on Standard ML are being developed at the University of Calgary, Canada and by International Computers Limited (ICL), England. The Calgary version is in the public domain and has been beta-released since November 1991.

HOL88 is between the original ML and the standard ML.

The user of HOL interacts with the system by typing in expressions in ML. The system evaluates each input expression and prints out its value and type. Some expressions may also have side effects, for example, creating a theory file. This read-evaluate-print loop is known as the *top-level*.

Throughout this thesis, segments of ML programs and ML expressions are displayed typeset in typewriter font. In this section, meta-variables typeset in *italic* are used to stand for parts of an ML expression.

ML types Every expression in ML possesses a type. An expression may possess many types, in which case it is said to be *polymorphic*. Polymorphic types contain type variables whose names are strings of one or more 'a' characters optionally followed by a number or an identifier. For example, *a*, *aa*, *a2* and *aaafoo* are all legal type variables. The type of the expression is printed after the expression separated by a colon (:).

There are four basic types in ML: *void* which contains a single object denoted by (); *int* which stands for integers, such as 1, 2, 3 and so on; *bool* which stands for boolean values *true* and *false*; and *string* which stands for ASCII character strings which are enclosed by a pair of single quotes (') like this 'This is a string'. There are three special infix type constructors: *#* constructs a cartesian product, also known as a *pair*; *+* constructs a disjoint sum and *->* indicates a function type. All these type constructors are associated to the right, i.e., $(\text{int} \# \text{int} \# \text{bool})$ is equivalent to $(\text{int} \# (\text{int} \# \text{bool}))$. There is a pre-defined type constructor *list* which takes a single argument. For example, $(\text{int})\text{list}$ is the type for integer lists

and $(*)list$ is a polymorphic type standing for a list whose elements may be of any type.

The ML type checker uses a set of rules to check the type of every input expression before evaluating it. This strict type checking has its roots in the original ML language. It is important, among other reasons, because abstract types are used to represent terms and theorems of the logic and the values of the latter cannot be created arbitrarily. Incorrectly typed expression will cause an error message to be printed and no evaluation will be performed.

Expressions ML expressions consist of constants, variables, function applications, lambda expressions, conditionals, local declarations and exceptions.

Constants of the basic types are already described above. Constants of compound types can be expressed using constructors and basic constants. Since lists and pairs are used very often, a special syntax is provided for inputting expressions of these types. Elements of a list can be enclosed in a pair of brackets and each is separated by a semicolon. For example, $[1;2;3;4]$ is a list containing four integers. A pair is enclosed in parentheses and the elements are separated by a comma. The parentheses of nested pairs can be ignored, leaving the commas to act as right-associative operators. For example, $(1,2,true)$ is a pair of the type $(int * int * bool)$.

Names of *variables*, or *identifiers*, can be either a sequence of alphanumeric characters starting with a letter or a special symbol chosen from a list which can be programmed by the user. Identifiers of the second form will not be used in this thesis. An identifier can be bound to any value, for example, a HOL definition or

a theorem. All ML identifiers mentioned in the subsequent chapters will be typeset in typewriter font.

Function applications have the form $e_1 e_2$ where e_1 is an expression which must be evaluated to a value possessing a function type and e_2 must be an expression possessing a type which is an instance of the domain of e_1 . For example, the pre-defined function `hd` has the type $(\ast)list \rightarrow \ast$. The expression `hd[1;2;3;4]` is a well typed function application. The type of this expression will be `int` which is obtained by substituting the type of the argument, namely `(int)list` into the function type, the type variable \ast is then instantiated by `int`.

A *lambda expression* has the form $\backslash x. e$ where the ASCII character `\` is used to approximate the lambda symbol (λ) used in conventional mathematics. The evaluation of any lambda expression always yields a function value whose type is $t_{y_1} \rightarrow t_{y_2}$ where t_{y_1} is the type of x and t_{y_2} is the type of e .

Conditionals in ML have the form `if e then e' else e''`. When a conditional is evaluated, the expression e , which must possess type `bool`, is evaluated first. If the result is `true`, e' is evaluated, otherwise e'' is evaluated. The value of either e' or e'' will become the value of the conditional. The `else e''` part is optional. If this part is omitted and e is evaluated to `false`, the unique value of type `void` is returned.

Local declarations have the form `d in e` where d is a declaration as described below. The scope of this declaration is the expression e .

Exceptions are a special class of expression. Their purpose is to trap errors. They take the form `failwith e`. When an exception is encountered, the expression e is evaluated whose result must be of type `string`, then a failure is generated.

Declarations ML declarations take one of the following forms:

`let $x = e$`

`let $f\ x_1 \dots x_n = e$`

`letrec $f\ x_1 \dots x_n = e$`

The first declares a variable, then e is evaluated and the resulting value is bound to x . The second declares a function with name f and formal arguments $x_1 \dots x_n$. The last form is the same as the second except that the declared function is recursive.

2.2 Overview of the HOL Logic

In classical propositional logic, each proposition can be either true or false, but not both. There are a number of logical connectives to combine simple propositions to form more complex ones, such as negation(\neg), conjunction(\wedge), disjunction(\vee), implication(\supset), and so on. This can be regarded as *zero-order logic*.

In predicate logic, there is an infinite set of variables and, for each $n \geq 0$, a set of n -place predicates. There are quantifiers, such as *forall*(\forall) and *there exists*(\exists) to quantify variables. This can be regarded as *first-order logic*.

In the HOL logic, variables can range over functions and predicates, a function can take another function as its argument and can deliver a function as the result of applying it. Such a function is called a *higher-order function* or *functional*. One can talk about 'for all function f ' and so on. Hence, it is a *higher-order logic*.

Expressions of the HOL logic are *terms*, and they are represented in ML by values of type `term`. They are usually input using the mechanism known as *quotation* in which a logical expression is enclosed by a pair of double quotes, like " $A \wedge B$ ". To

the HOL system, " $A \wedge B$ " is an ML expression of type **term**, and it denotes a logical term meaning 'the conjunction of A and B '.

Every expression in the HOL logic belongs to a type which can be thought of as a set of objects having certain common properties. This type is known as *logical type* in contrast to the ML types of ML expressions. For example, the logical expression " $A \wedge B$ " has logical type **bool** which stands for boolean. All logical types are typeset in **typewriter** font prefixed by a colon (:) as in the above example. Logical types are represented in ML by an abstract type **type**. The ML function **type_of** takes a term and returns its logical type. This may seem confusing. The HOL session below may help to clarify the difference between logical types and ML types.

```
# let t = "A ∧ B";;  
t = "A ∧ B" : term  
  
# type_of t;;  
":bool" : type
```

In the above example, the hash sign (#) is the HOL system prompt and the double semi-colon (;;) terminates each input expression. The line between them is the input typed by the user. The next line is the system response. In the first line, the user declare an ML variable **t** and binds the logic expression " $A \wedge B$ " to it. HOL responds with the value of this variable and its type. In the second input line, the user applies the function **type_of** to **t**. HOL responds with the logical type of the term.

2.2.1 HOL types

Logical types can be one of the following:

- a **type variable** which stands for an arbitrary set of objects. This is also known as *polymorphic type*. Names of logical type variables are constructed following the same rules as ML type variables.
- an **atomic type** or constant type which stands for a fixed set of objects. Some of the pre-defined atomic types are `:bool` for booleans, `:num` for natural numbers, `:one` for a set containing only a single element and `:inf` for an infinite set.
- a **function type** which is written as $:t_{y_1} \rightarrow t_{y_2}$. This stands for a set of functions whose *domain* is the set t_{y_1} and whose *range* is the set t_{y_2} .
- a **compound type** of the form $(t_{y_1}, \dots, t_{y_n})op$ where t_{y_1}, \dots, t_{y_n} are types, known as *argument types*, and *op* is a type operator. *op* is said to be of *arity* n since it takes n types as its arguments. A compound type stands for the set resulting from applying the type operator *op* to the sets denoted by t_{y_1}, \dots, t_{y_n} . `list` is a pre-defined type operator of arity 1, and `prod` is a pre-defined type operator of arity 2 which stands for cartesian products, also known as pairs. For example, `:(num)list` is the type for lists of natural numbers, and `:(num,bool)prod` is the type for pairs whose first elements are natural numbers and whose second elements are booleans. Since pairs are used frequently, a special syntax is provided for the type operator `prod`; an infix `#` can be used, for example, the type `:(num,bool)prod` is usually written as `:num # bool`.

Function types and constant types can be considered as special cases of general compound types. A function type $:t_{y_1} \rightarrow t_{y_2}$ is equivalent to $:(t_{y_1}, t_{y_2})\text{fun}$. Constant

Kind of term	HOL notation	Standard notation	Description
Variable	$v : ty$	v_σ	variable of type σ
Constant	$c : ty$	c_σ	constant of type σ
Combination	$t_1 \ t_2$	$t_1 t_2$	apply the function t_1 to the argument t_2
Abstraction	$\lambda x. t$	$\lambda x.f$	λ -abstraction

Table 2.1: HOL primitive terms.

types are type operators of arity 0.

A polymorphic type ty containing type variables $tyvar_1, \dots, tyvar_n$ can be subjected to a *simultaneous substitution* by the types ty_1, \dots, ty_n . The resulting type ty' is called an *instance* of ty . For example, $(bool, num)prod$ is an instance of the type $(tyvar_1, tyvar_2)prod$.

2.2.2 HOL Terms

Well formed expressions in HOL are called *terms*. Unlike predicated logic, there is no separate syntactic class for formulae; their roles are played by terms of type $:bool$. Every term belongs to a type and denotes an element of the set denoted by that type. The HOL system quotation parser attempts to deduce the type of a term when it is input in a quotation. Sometimes, there is not enough information for the parser to work out the type, in such case, explicit type information can be attached to the term or any part of it. For example, in $"x : bool"$, the variable x is specified to be of type $:bool$.

Kind of term	HOL notation	Standard notation	Description
Truth	τ	\top	<i>true</i>
Falsity	F	\perp	<i>false</i>
Negation	$\sim t$	$\neg t$	<i>not t</i>
Disjunction	$t_1 \vee t_2$	$t_1 \vee t_2$	<i>t₁ or t₂</i>
Conjunction	$t_1 \wedge t_2$	$t_1 \wedge t_2$	<i>t₁ and t₂</i>
Implication	$t_1 \Rightarrow t_2$	$t_1 \supset t_2$	<i>t₁ implies t₂</i>
Equality	$t_1 = t_2$	$t_1 = t_2$	<i>t₁ equals t₂</i>
Conditional	$(t \Rightarrow t_1 \mid t_2)$	$(t \rightarrow t_1, t_2)$	<i>if t then t₁ else t₂</i>

Table 2.2: HOL infix terms and special constants.

Kind of term	HOL notation	Standard notation	Description
\forall -quantification	$\forall x. t$	$\forall x. t$	<i>for all x : t</i>
\exists -quantification	$\exists x. t$	$\exists x. t$	<i>for some x : t</i>
ϵ -term	$\epsilon x. t$	$\epsilon x. t$	<i>an x such that: t</i>

Table 2.3: HOL binders.

Table 2.1 lists all kinds of primitive terms in HOL. All terms can be constructed from these primitives. Some constants are given special syntactic status of *infix* or *binders*. The pre-defined infix constants are listed in Table 2.2 and binders are listed in Table 2.3.

Within a quotation, an expression of the form $\text{'}(t)$ is called an *anti-quotation* where t must be an ML expression of type *term* or *type*. Such an expression evaluates to the value of t .

In subsequent chapters, HOL terms will be displayed and typeset in typewriter font using the quotation mechanism, i.e., always enclosed by a pair of double quotes ("), such as

`"A /\ B = B /\ A"`

When a term is referred to in running text, the logical constants are typeset in Sans Serif font while the standard notations listed in the tables above will be used for variables and special constants.

2.2.3 Theories

The result of a session with the HOL system is an object called a *theory*. A HOL theory is very similar to a logician's theory. Like a logician's theory, a HOL theory contains types, constants, definitions and axioms. The most important difference is that a HOL theory also contains an explicit list of theorems which have been proved from the axioms and definitions using the theorem prover while a logician's theory implies all theorems (often infinitely many) that could be proved. Therefore, all HOL theorems mentioned in subsequent chapters have actually been proved.

A HOL theory is stored in a number of files called the *theory files*. Each theory

file contains some types, constants, axioms and theorems, together with pointers to other theory files called its *parents*. The collection of reachable files is called the *ancestry* of the theory. When the HOL system starts, the initial theory is the theory HOL. The ancestry of the theory HOL contains all types, axioms, constants and theorems of the HOL logic. All new theories created during a HOL session are extensions of the theory HOL, either directly or via some other theories such as those provided as libraries. The names of theories and libraries are typeset in typewriter font.

A theory can be extended in the following ways:

- by **constant definition** which introduces new constants by specifying formulae to determine them uniquely;
- by **type definition** which introduces new types or new type operators by specifying a non-empty subset of an existing type and proving that the new type is isomorphic to this subset;
- by **constant specification** which introduces new constants which satisfy arbitrary given consistent properties. The constants may not be uniquely determined. There is an ML function for this purpose.

All these extensions to theory are known as *definitional extension*. Theories created solely by definitional extensions are called *definitional theories*. Since the new constants and types are defined in terms of properties of existing ones, the extended theory is consistent if the original theory is. All the theories described in the subsequent chapters are definitional theories, and since they are extensions of the HOL theory, they are consistent. Only the first two methods of theory extension have

been used in developing these theories.

There are several pre-defined ML functions in HOL for defining constants. The result of calling these functions is an equational theorem characterizing the newly defined constant. The session below defines a new constant **GRAPH** using the function `new_definition`.

```
# let GRAPH_DEF = new_definition('GRAPH_DEF',
# "GRAPH ((V:('Vertex)set),(E:('Edge)set)) =
# {a. a IN E ==> (((a_src a) IN V) /\ ((a_des a) IN V))}";
GRAPH_DEF =
|- (V E. GRAPH(V,E) = {a. a IN E ==> (a_src a) IN V /\ (a_des a) IN V})
```

The resulting theorem is stored in the current theory with the name **GRAPH_DEF**. By convention, all definitional theorems are named with the suffix `_DEF`. In this dissertation, new constant definitions are presented in the following form:

HOL Definition 1 (GRAPH_DEF)

```
"GRAPH ((V:('Vertex)set),(E:('Edge)set)) =
{a. a IN E ==> (((a_src a) IN V) /\ ((a_des a) IN V))"
```

The string following the definition number is the name of the definition. The term characterizing the new constant is printed in the HOL input notation in **typewriter** font.

A type definition package is provided to allow concrete recursive types to be defined automatically [48]. It provides an ML function `define_type` which accepts a simple type specification language in the form of:

$$type = C_1 t_{y_1}^1 \dots t_{y_1}^{k_1} \mid \dots \mid C_m t_{y_m}^1 \dots t_{y_m}^{k_m}$$

where *type* is the name of the new type and C_i are the type constructors. The results of defining a new type are

- to formally define *type* as a type in the current theory;
- to make appropriate constant definitions for the constructors C_i ;
- to automatically prove a theorem which characterizes the newly-defined type.

The type definition package also provides a set of functions to automatically prove theorems about the basic properties of the type. These functions are:

`prove_constructors_one_one` which proves a theorem stating the constructors are one-one functions;

`prove_constructors_distinct` which proves a theorem stating that, if more than one has been defined, the different constructors produce different objects in the type;

`prove_cases.thm` which proves a theorem stating that any object of the type is produced by one of the constructors;

`prove_induction.thm` which proves a theorem stating the structural induction principle of the type.

The session below defines a new type `:Tc1r` with a constructor `TC1R`. The type characterization theorem is stored with the name `Tc1r_Axiom`; by convention, all type characterization theorems are named with suffix `_Axiom`.

```

Slet Tc1r_Axiom = define_type 'Tc1r_Axiom'
'Tc1r = TC1R num (num->Tstate)';;
Tc1r_Axiom = |- !f. ?! fa. !n f'. fn(TC1R a f') = f n f'

```

In this dissertation, new type definitions are presented in a format similar to that of constant definitions. The string specifying the new type is enclosed in a pair of single quotes('), which resembles the input syntax required by the type definition package. Below is the definition of `:Tc1r` shown in such a format.

HOL Definition 2 (Tc1r.Axiom)

```
'Tc1r = TC1R num (num->Tatata)'
```

2.2.4 Theorems

A *theorem* is the result of a proof. In a more formal sense, a *proof* is a list of pairs $[(\Gamma_1, i_1), \dots, (\Gamma_n, i_n)]$ known as *sequents* in a *deductive system*, and a theorem is the last element of this list. The first component Γ_i of a sequent is a set of formulae called the *hypotheses* or the *assumptions* and the second component i_i is a single formula called the *conclusion*. Each sequent in a proof is either a theorem that has been proved earlier or is derived from other theorems following some rules known as the *inference rules*.

Theorems in the HOL system are represented by values of the ML abstract type `thm`. There is no way to construct a value of type `thm` except by carrying out a proof. In this way, the ML type system protects the HOL logic from the arbitrary construction of a theorem, so that every computed value of the type representing theorems is a theorem.

The HOL system prints values of type `thm` in a special way: it prefixes the conclusion with `|=` which resembles the turnstile (\vdash) in the conventional mathematical notation for theorems. For example, the theorem asserting the symmetry of addition is printed as:

```
|= !m a. (m + a) = (a + m)
```

There are five axioms in the HOL logic, which are the only pre-defined values of type `thm`. All other theorems follow from them.

Once a theorem has been proved, it can be saved in the current theory. Every

theorem is identified by two strings: the name of the theory file in which it is stored and the unique name of the theorem. Theorems stored in any file reachable from the current theory can be loaded into the current HOL session and bound to an ML identifier. Theorems can be loaded automatically if autoloading has been set up. In this case, a theorem is loaded whenever its name is first mentioned in the input, and it is bound to an ML identifier of the same name. For example, the 'symmetry of addition' theorem is stored in the theory `arithmetic` and has the name `ADD.SYM`. When this name first appears in an expression, the theorem will be loaded and bound to the ML identifier `ADD.SYM`.

In this dissertation, all theorems are printed in conventional mathematical notation rather than the raw output from the HOL system as shown above to improve the readability. The format is similar to that used for definitions. An example is shown below:

HOL Theorem 1 (`G.UNION.SYM`)

$$\vdash \forall G_1 G_2. G_1 \text{ G.UNION } G_2 = G_2 \text{ G.UNION } G_1$$

The string after the theorem number is the name under which the theorem is stored. The same ML identifier will be bound to the theorem if it is automatically loaded.

The conversion from the system output format to the format given above is performed by a formatter developed by the author. The formatter is organized as a library named `latex-hol`. It consists of a set of ML functions which takes a theorem or a whole theory and generates text in \LaTeX format. The text can then be typeset using the \LaTeX typesetting system. The implementation of the formatter is based on the HOL pretty printer library `pretty`. Details of how to use the formatter and

its implementation can be found in [67].

2.3 Theorem proving in HOL

Proofs can be carried out in two different ways: *forward proof* and *goal-directed proof*.

A forward proof starts with an existing theorem and inference rules are applied successively to transform this theorem into a sequence of new theorems until the desired one is reached. A goal-directed proof sets up a goal which has exactly the same form as the desired theorem, then tactics are applied to decompose it into a list of subgoals, and this process continues until all subgoals can be solved.

2.3.1 Forward proof

Inference rules are used to transform a theorem when carrying out forward proofs. They are implemented in HOL as ML functions which deliver a theorem. These functions take one or more theorems and possibly other arguments depending on the meaning of the inference rules. There are eight primitive inference rules in HOL. Since there is no primitive constructor for the values of type `thm` in ML, calling these functions is the only way to create a theorem. The primitive inference rule *Modus Ponens*, for example, is represented by the ML function `MP`. It takes two theorems as its arguments: the first should be an implication and the second should be a theorem matching the antecedent of the implication. It returns a theorem in the same form as the conclusion of the implication.

There is a comprehensive set of pre-defined ML functions implementing *derived* inference rules. These functions are defined in terms of the eight primitive rule

functions. Each of them combines a number of steps of applying the primitive rules, and thus providing a set of more useful tools. Users of HOL can also define their own functions implementing derived inference rules for their special needs.

A *conversion* in HOL is a rule which maps a term to a theorem stating the equality of that term to some other term. The theorem produced by a conversion is often used to convert the whole or part of a formula — i.e., rewriting or substitution. Conversions are also represented by ML functions returning a theorem. For example, the conversion `bool.EQ_CONV` takes a term for the form " $b_1 = b_2$ " and returns one of the following theorems:

- $\vdash (b_1 = b_2) = \text{True}$ if b_1 and b_2 are identical boolean terms, or
- $\vdash (b_1 = b_2) = \text{False}$ if each of b_1 and b_2 is `True` or `False` but different, or
- $\vdash (b_1 = b_2) = b_2$ if b_1 is `True`, or
- $\vdash (b_1 = b_2) = b_1$ if b_2 is `True`.

There are a number of higher-order functions for combining conversions to form more complex ones. For example, `conv1 THEN conv2` is a conversion formed by the function `THEN`. When this expression is evaluated, the conversion `conv1` is performed first, then the conversion `conv2` is carried out.

2.3.2 Goal directed proof

The forward proof style is rather unnatural, and is too 'low level' for many applications. The goal directed proof style constructs a proof by organizing it into a tree in which each node is a subgoal and each edge is a tactic. The tree is traversed twice, the first is from the root representing the original goal to the leaves representing the final (trivial) subgoals, the second is from the leaves back to the root. In the

first pass, tactics are applied and subgoals are generated, and in the second pass, a proof is computed from the theorems achieving the subgoals to yield the theorem achieving the original goal. This is only a conceptual view of the goal-directed proof style. No tree is actually created in HOL, but there is a subgoal package which manages all the proof searching efforts. It provides functions for the user to set up a goal and then to apply tactics. The idea of using tactics in goal-directed proofs originated from Milner and was first implemented in Edinburgh LCF [35].

A *tactic* is an ML function. When applying to a goal, a tactic reduces it to

1. a list of subgoals, and
2. a *justification function* mapping a list of theorems to a theorem.

The subgoal package keeps track of the justification functions and combines them in the correct order to compute the final theorem that achieves the goal. There is a comprehensive set of tactics provided by the HOL system. Tactics can be combined to form more complex ones using *tacticals*. The user can define special tactics for his application using the existing ones. In Chapter 6, the proof of a theorem asserting a property of networks will be described in detail to illustrate the process of proof searching in the goal-directed proof style.

Chapter 3

Principles of Railway Signalling

This chapter gives an introduction to the principles of railway signalling and the state-of-the-art technology in automatic interlocking systems so that readers not familiar with the subject can understand the following chapters.

The primary function of a railway signalling system is to maintain the safe operation of trains over the track network and to protect human beings from injury and equipment from damage. In addition to this, the system should allow efficient operation of trains so that the maximum capacity can be obtained.

This chapter describes the signalling equipment, the principles of railway signalling and the state-of-the-art in automatic signalling systems based on the current practice of British Rail [54]. The description is rather general and provides a viewpoint for the research described in later chapters. Many important issues of railway signalling which are not within the scope of this research are ignored. Most of the concepts described here are also applicable to other railway authorities.

Section 3.1 describes the basic functions of individual components of signalling equipment. The central concept of signalling systems — *interlocking* — and the

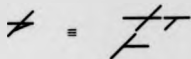


Figure 3.1: A slip crossing and its equivalent atomic parts.

operation of the complete system are explained in Section 3.2. The last section of this chapter presents the current technology of signalling systems in British Rail.

3.1 Railway track and signalling equipment

A signalling system consists of many kinds of component. These components can be grouped into two classes: track components and signals. The first class forms the railway track network, the *permanent way*, and the second is the means of controlling the train movements over the track network.

3.1.1 Track components

For the purposes of this research, the property of the track components which is of interest is their topology, that is, how they are interconnected to form a track network. Based on this consideration, the track components have been divided into the following four classes: buffers, tracks, points and diamond crossings. They are called *parts*. Table 3.1 below shows a schematic drawing of these parts together with a brief description.

There are other, more complex track components in real networks, such as the single slip crossing. It can be considered as a compound of a simple diamond crossing with a pair of simple points connected to either end of it. This is illustrated in





	buffer — the dead end
	plain track — simple track sections without any branches and crossing
	point (switch) — a branching point which can be set at either 'NORMAL' or 'REVERSE' positions to route train to different directions
	diamond crossing — an intersection where two routes are crossing each other

Table 3.1: Track components.

Figure 3.1 with the single slip crossing shown on the left-hand side and the decomposition to atomic parts on the right. The four simple components listed above are atomic. All complex track components can be built up by combining appropriate atomic parts. A complete track network can be formed by connecting the required track components together and placing signals at the appropriate locations.

The primary means of detecting the presence of a train is by the use of track circuits. A conceptual view of the operation of the circuits is illustrated in Figure 3.2. A voltage is applied to the two rails of the track. This can be detected by a sensor to indicate that the track section is 'CLEAR'. When a train is present, its wheels bridge the circuit reducing the voltage between the rails, so that the output is changed to 'OCCUPIED'. The design of a track circuit is fail-safe, in that it will always give an 'OCCUPIED' output when it is faulty.

In a fixed block system, the track network is divided into sections. Each section

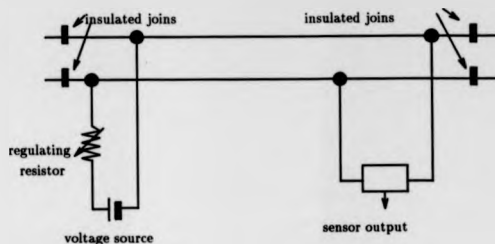


Figure 3.2: The principle of track circuit.

is controlled by a signal and may be occupied by at most one train except in certain special operations, such as the coupling of an engine. Track sections are electrically insulated from each other. In a simple layout, each section usually has its own associated track circuit, and consists of only a single part, for example, a plain track. In a complex layout, such as a busy terminal station, a section may contain several track circuits, each circuit spanning several parts.

The point where the adjacent sections meet is called a *join*. Since the track is characterized by different kinds of part, the term *join* is also used to mean the meeting point of two adjacent parts. In practice, there are several types of join:

- conducting join — which is the join between two parts that share the same track circuit;
- insulated join — which is the join between two track circuits;



Figure 3.3: An overlap.

- overlap join — which is also formed between two track circuits, but in addition, one of the circuits is a special section of track, known as an *overlap*.

An *overlap* is a short section ahead of a *stop signal*, whose function is to protect a train against overrunning the signal in adverse conditions. As illustrated in Figure 3.3, the section between joint *j2* and *j3* is an overlap which protects a train from overrunning signal *S11*, and thus *j3* is an overlap joint.

3.1.2 Signals

Like track components, there are several kinds of signals. Their names and functions are listed below:

- main signal** — gives instructions to the normal running traffic;
- junction indicator** — provides information at the entry to a branch so that a diverted train may slow down;
- subsidiary signal** — (always associated with a main signal) authorizes the driver to pass the main red aspect and draw ahead to stop short of any obstructions;
- shunting signal** — gives instruction for slow movements into or out of sidings etc.

Sometimes, a combination of several types of signals is installed on a single signal post, for example, a main signal and a junction indicator are often combined at the

entry to a branch line.

According to the number of different aspects they can display, main signals can be of 2, 3 or 4-aspect. For 2 and 3-aspect types, there are *stop signals* and *repeaters*. The repeaters are used where earlier warning is required of the aspect of a stop signal. A 2-aspect stop signal can display RED and GREEN while a 2-aspect repeater can display YELLOW and GREEN. A 3-aspect signal can display RED, YELLOW, GREEN. A 4-aspect signal can display a 'DOUBLE YELLOW' aspect in addition to all aspects of the 3-aspect signal. The use of different number of aspects depends on the traffic density, the headway and the length of the track sections. For example, in Figure 3.3, S11 is a 2-aspect stop signal with an overlap protection and S10 is a 2 aspect repeater.

At any time, a signal is in either of the two states: 'ON' or 'OFF'. A train must not pass a signal which is in the 'ON' state; this is shown as the RED aspect. All other aspects are said to be 'OFF'. All aspects of main signals employ double filament bulbs. The auxiliary filament is switched into use automatically as soon as the main filament is broken. When this happens, an indicator on the control panel is illuminated to warn the signalman. A signal will not be taken as showing a particular aspect simply because it is selected. There is *proving circuit* built into the signal which checks whether the selected aspect is drawing current, that is the bulb for the selected aspect has at least one filament illuminated. In the case in which both filaments of the selected aspect are broken, the signal is said to be *faulty*.

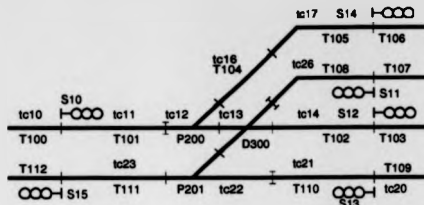


Figure 3.4: An example of track layout: a double left-hand junction.

3.2 Signalling principles

Signalling systems provide two types of controls to the traffic: the first is route control, which is achieved by setting the appropriate points; and the second is speed control which is achieved by setting the aspects of the appropriate signals. All these controls are operated from a central location — the signal control centre.

Each control centre may supervise an area of up to hundreds of kilometres of tracks, which consists of many points and signals. The track network is organized into a number of *routes*. Each route starts from a signal, the *entry signal*, and usually ends at another signal, the *exit signal*. If two routes share at least one part, they are said to be *conflicting*. To allow a train to pass through a route, it has to be set up and *proved*. This means that the conditions which provide a safe passage through the route have to be satisfied. For example, Figure 3.4 shows a layout in the vicinity of a left-hand double junction. Suppose that the route from signal *S10*

to *S12* is to be set up. This requires:

- all the track circuits along the route, namely *tc11*, *tc12*, *tc13* and *tc14*, to be 'CLEAR';
- the point *P200* to be set to *NORMAL* and detected at the correct position;
- the exit signal *S12* is not faulty;
- the entry signal *S11* of the conflicting route from this signal to *S15* is turned 'ON' and proved to be alight;
- the track circuit along the conflicting route up to the point of conflict, namely *tc26*, must be 'CLEAR'.

After all of these conditions are satisfied, the entry signal to the route can then be turned 'OFF', and the route is said to be 'set'.

Central to the operation of the signalling system is the concept of interlocking. For example, the route locking is in operation after the above route is set. This means that the satisfied conditions should not be destroyed by subsequent operation of points or setting up a conflicting route. This is important because the incoming train has been given permission to proceed and if the entry signal is changed to 'ON' unexpectedly, the driver may not be able to stop short of it and dangerous situations could arise.

Traditionally, these operation conditions are expressed in a tabular form, the *control tables*. The control tables have a well-defined syntax and semantics for the interlocking functions, and are very well understood by signalling engineers. A control table for the example route above is shown in Table 3.2.

ROUTE	TRACK CIRCUITS		REQUIRE POINTS		SIGNALS	
	CLEAR	OCCUPIED	NORMAL	REVERSE	ALIGHT	ON
S10512	tc11, tc12		P200		S12	
	tc13, tc14					
	Protect from conflicting traffic					
	tc26					S11

Table 3.2: An example of control table.

3.3 Solid-state interlocking

In the early days of railway signalling, interlocking was achieved by the mechanical interlocking frame. When electromechanical technology was adopted, relay circuits implemented the interlocking function and are still in widespread use. Since its advent in the late 70's, microprocessor technology has influenced every branch of engineering and railway signalling is certainly no exception. However, the application of microcomputer control technology in signalling systems is rather conservative. This is because of the strict safety assurance required in such systems. Due to the unpredictable failure modes of a complex microcomputer control system, microprocessors were only used in non-vital functions in the early attempt of applying computer in signalling controls.

The first use of microprocessor in control of vital safety functions in the U.K. was the *Solid State Interlocking* project of British Rail Research[20][21] which started as soon as the first 16-bit microprocessors became available. The aim of the project was towards the eventual replacement of present day electromechanical signalling

interlocks'[22]. At the same time, the new system should not alter any signalling principles and the appearance and behaviour of the system so far as the operator is concerned. The result is the now highly-acclaimed SSI system, which has been adopted for all new signalling installations in British Rail.

3.3.1 Features of the SSI

The SSI system can be divided into three parts:

- one or more microcomputer interlockings;
- a control panel;
- a maintenance terminal.

The heart of the computer interlocking is the interlocking processors. There are three processors which operate as a *triple modular redundancy*(TMR) system to achieve the strict requirements of reliability and fault tolerance. These perform all the safety-critical logical operations of the signalling controls. The commands to the equipments, such as point machines and signals, are transmitted to specialized interface units at the trackside via a duplex data highway. The data on this highway are encoded in two levels to withstand the severe electromagnetic interference encountered with electric traction and to maintain the high overall integrity required. Two panel processors perform the non-safety tasks of servicing the control panel which can be either the conventional mosaic push-button type or new style Visual Display Units (VDU) type. A diagnostic processor provides information to the maintenance terminal which is used by the technician to monitor the performance of the system.

The software controlling the interlocking processors is designed to be data driven. Since every signalling scheme has a different configuration, it will be very inefficient, if not impossible, to design, implement and verify a special version of the control program for each installation. The arrangement of the signals, the track layout and the rules for controlling them are encoded in a data base. The standard software is based on the concept of the control cycle: in each cycle, the system processes one incoming message and generates one control command. During the cycle, it updates a set of variables representing the current state of the network, consults the database for the applicable rules, and derives the correct control commands.

Since the software is safety-critical, it is subjected to rigorous testing and validation processes to ensure that it is logically correct and faithfully implements the specification. The software development and validation processes were described in [60]. The basic principles are to be disciplined in the development stage and to be rigorous in the validation stage.

Very strict rules are applied to the design and development of the software. The programs are highly modular and well structured. The use of interrupts is excluded in favour of simple looping and polling. Data flow between modules is made explicit. High quality documentation has been produced which contributes to the correctness of the software and simplify the validation process. Several well-established techniques are used to validate the software: *functional analysis* checks the correspondence between the requirement specification and the software; *structural analysis* checks the programming logic; *information flow analysis* ensures the correct data passing between modules; *modular analysis* confirms the correctness of each module; and *semantic analysis* proves the software correct by logical deduction.

The whole software development process, from specification through design to validation, makes certain assumptions about the environment within which the final program will execute. These include the behaviour of the hardware, the equipment which under the program control. Only system testing will uncover any misconception on the reality. Therefore, extensive system testing has to be carried out, both in an environment simulated in a laboratory and in the real field environment.

3.3.2 The design procedures

Briefly, the procedures of designing and implementing a signalling scheme using SSI [50] is as follows:

1. specify the required track and signalling layout;
2. produce *control tables*;
3. generate *geographic data*;
4. test the geographic data on a simulator;
5. install the SSI with EPROMs containing the geographic data.

One of the major task of the railway signalling engineer is to produce the control tables required for a particular track and signalling layout. The geographic data are then extracted from this table and written in a special purpose high-level programming language. This program is subsequently compiled into data object codes and installed in the SSI. The process of producing control table and geographic data is being carried out manually because it requires skill and experience. The research described in later chapter shows a possible way of automating these operations.

3.3.3 The benefit and the future

The major benefits of the SSI system are its flexibility, good maintainability and cost-effectiveness. The flexibility is due to the modular organization of the system which provides an easy path to future evolution for both the interlocking itself and other systems it interfaces with. For example, the Integrated Electronic Control Centre (IECC)[61] is the new human interface built on top of the interlocking. It replaces the conventional mosaic push-button control panel by track balls, keyboards and VDU. The modular system also contributes to other benefits, the bulk of the cost saving being achieved by the dramatic shrinkage of the physical dimensions of the new equipment.

The flexibility of the SSI system also opens up the possibility of higher order control systems. The Automatic Route Setting (ARS) is one of such systems. It is interlinked with the Signalling and Information Networks. It is informed of the pre-planned schedule, the location of the trains, and it learns from the SSI the real-time state of the signalling and track occupation. From this information, it can deduce the required route and whether there are any conflicts. It can then calculate the optimal routing strategy and translate it into commands to the SSI.

Besides the SSI system in the U.K., railway authorities in other countries have also developed computer controlled signalling systems, such as the ERILOCK systems in Swedish Railway Administration[13], the SMILE systems in Japanese National Railways[2], and so on.

Since the SSI is the key element in providing interlocking, a rigorous formal approach to the design and implementation is necessary and desirable. The research

described in the later chapters was carried out with the aim of helping railway signalling engineers and designers to improve the integrity and correctness of interlocking systems based on the use of microprocessors and real-time software.

Part II

Theories

In order to apply formal methods in signalling systems, a formal model of the railway track network and signalling equipment must first be established. This model is an abstract view of the railway track network, based upon which, specifications of interlocking system can be developed. This part presents such a formal model expressed in Higher-Order Logic. It is organized into several theories which form a hierarchy shown on the next page.

In the hierarchy, the *sets* theory is one of the HOL system libraries. The theories below the dashed line are developed by the author.

On the left-hand side are the theories describing the mathematical structures, namely *graphs* and *paths*, which are used to model the network. The main theories *graph* and *path* contain definitions and theorems about these mathematical structures, while the auxiliary theories *func* and *elist* contain some lower level definitions and theorems about functions and lists which support the main theories. These theories are described in Chapter 4.

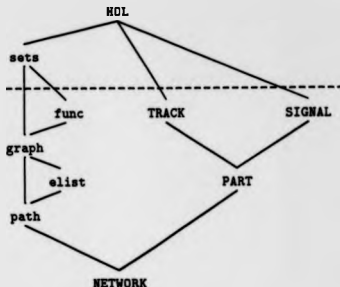


Figure II: The hierarchy of theories.

On the right-hand side of the hierarchy are the theories modelling the railway equipment. The theory **TRACK** models track components, and the theory **SIGNAL** models signals. The **PART** theory combines individual components to provide a uniform interface to the network model. These theories are described in Chapter 5.

The top level of the model is the theory **NETWORK** which characterizes track networks. Some basic properties of such networks have been proved. This is described in Chapter 6. The complete listings of each theory can be found in Appendix A and the ML sources for creating these theories are listed in the Appendix B.

Chapter 4

The mathematical foundation

This chapter describes the HOL theories graph and path and two auxiliary theories func and alist. These theories contain the abstract mathematical structures for modelling the network.

Graph theory is a very large branch of mathematics, and it has found applications in many diverse fields. The theory described in this chapter is only a first attempt at expressing a small portion of the conventional graph theory in Higher-Order Logic. The main criteria for deciding what to include in the theory are the requirements of modelling the track network.

The definitions and theorems about graphs and paths are organized into two main HOL theories, graph and path, and two auxiliary theories, func and alist. The graph theory contains definitions of labelled directed graphs, several basic relations of vertices and edges, and basic operations on graphs. Some properties of graphs and related operations have been proved and the theorems are stored in this theory. The path theory contains definitions of walks, trail and paths, and basic operations on them. Similarly, some basic properties in the form of theorems are stored in this theory. The func theory contains definitions and theorems about functions

some of which are used in the reasoning of graph isomorphism. The features used will be described when graph isomorphism is discussed. The *elist* theory contains some operations and facts about lists which are needed for reasoning about paths. It is described in Section 4.2.2.

The theories have been developed in a very general way so that they will be suitable to be used for other applications. The terminology and definitions adopted in these theories follow the convention found in many textbooks, such as [64] [65] [29]. Since there is not a library of graphs in the HOL system up to version 2.0, the theories as described below provide a starting point for building a comprehensive library of graphs. Thus, when applications requiring the use of graph arise, the library can be called upon without the repeated work of defining a graph.

4.1 The theory graph

4.1.1 The representation of graphs

First of all, graphs have to be represented by some structure in HOL. This representation should reflect the abstract properties of graphs, and should be general and flexible so as to be suitable for use in different applications. Based on these considerations, a type and a predicate have been chosen to represent graphs. The type is a pair of sets. The first element of the pair is the set of vertices which can be of any type, that is $\alpha::*$. And the second is the set of edges. Each edge is a triple containing the *source* vertex, the *destination* vertex and its own *label*. The vertex fields are of the same type of the elements of vertex set. The label field is of

a distinct polymorphic type `"**"`. For convenience, it is abbreviated in ML as¹:

```
let Vertex = ":" and
    Edge = ":( $\alpha$   $\beta$   $\gamma$   $\delta$   $\epsilon$ )" and
    Graph = ":( $\alpha$ )set  $\beta$  ( $\gamma$   $\delta$   $\epsilon$ )set";;
```

Thus, antiquotation can be used to make the subsequent HOL text much more readable. `"Graph"` is the polymorphic type for a general graph. Any particular instance of graph can be created by instantiating the types `" α "` and `"**"` with the required types for the vertices and for the labels of the edges.

The choice of this concrete representation follows most conventional definitions of graphs. However, not every object of the type `"Graph"` is a proper graph. A predicate is required to distinguish those which are graphs from others in the type. The definition of this predicate reads:

HOL Definition 1 (GRAPH_DEF)

```
"GRAPH ((V:('Vertex)set),(E:('Edge)set)) =
  (i.e.  $\alpha$  IN E ==> (( $\alpha$ _src  $\alpha$ ) IN V) /\ (( $\alpha$ _des  $\alpha$ ) IN V))"
```

where `α _src` and `α _des` return the first and second field of the triple representing the edges, respectively. This specifies that, to be a graph, the source and destination vertices of every edge in the edge set must be elements of the vertex set. This is the dominant abstract property of a graph.

Having this definition of a graph, we need to assert that there exists at least one graph, i.e., the theorem `GRAPH_EXISTS`.

HOL Theorem 1 (GRAPH_EXISTS)

$\vdash \exists G. \text{GRAPH } G$

¹ `new.type.abbrev` cannot be used since these are polymorphic types.

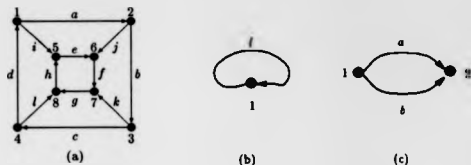


Figure 4.1: Examples of simple graph.

A trivial example of graph is the null graph $(\{\}, \{\})$. (A logical constant `NULL_GRAPH` is defined to be a null graph.) and a more interesting graph shown in Figure 4.1(a) can be written in HOL as

$$\begin{aligned} & \{(1, 2, 3, 4, 5, 6, 7, 8), \\ & \{(1, 2, a), (2, 3, b), (3, 4, c), (4, 1, d), (5, 6, e), (6, 7, f), \\ & (7, 8, g), (8, 5, h), (1, 5, i), (2, 6, j), (3, 7, k), (4, 8, l)\} \} \end{aligned}$$

As a consequence of this representation, all graphs are *directed*. This is because $(v_1, v_2, x) \neq (v_2, v_1, x)$ for all v_1, v_2 and x . However, it is still possible to represent an undirected graph using the same representation. Each edge of an undirected graph can be replaced by a pair of anti-parallel edges. Also, all graphs are *labelled*. To represent an unlabelled graph, the label field of the edges can be instantiated by the type `one`. Since there is only one object in this type, all the labels will be identical, hence, $(v_1, v_2, x) = (v_1, v_2, y)$ for all x and y .

4.1.2 Some basic definitions of graph

Two constants, VS and ES, are defined to access the vertex set and the edge set.

They are characterized by the theorems VERTICES and EDGES.

HOL Theorem 2 (VERTICES)

$$\vdash \forall V E. VS(V, E) = V$$

HOL Theorem 3 (EDGES)

$$\vdash \forall V E. ES(V, E) = E$$

A *loop* is defined to be an edge whose source vertex and destination vertex are identical. Applying the predicate HAS_LOOP to a graph G will yield true if and only if the graph G contains a loop. The graph shown in Figure 4.1(b) has a loop, the edge labelled l .

HOL Definition 2 (LOOP_DEF)

"LOOP (e: 'Edge) = (e_src e = e_des e)"

HOL Definition 3 (HAS_LOOP_DEF)

"HAS_LOOP G = ?(e: 'Edge). (e IN (ES G)) /\ (LOOP e)"

A graph is said to have *multiple edges* if and only if there is more than one edge which has the same vertex as its source and the same vertex as its destination. This property is expressed in the predicate MULTI_EDGE. The graph shown in Figure 4.1(c) has multiple edges, $(1, 2, a)$ and $(1, 2, b)$.

HOL Definition 4 (MULTI_EDGE_DEF)

"MULTI_EDGE G = ?(e1: 'Edge) e2.
 (e1 IN (ES G)) /\ (e2 IN (ES G)) /\
 ~(e1 = e2) /\ (e_src e1 = e_src e2) /\ (e_des e1 = e_des e2)"

A *simple* graph is defined to be a graph containing neither loops nor multiple edges. A *finite* graph is a graph whose vertex set and edge set are both finite.

HOL Definition 5 (SIMPLE_GRAPH_DEF)

```
"SIMPLE_GRAPH (G:~Graph) =
  (GRAPH G) /\ ~(HAS_LOOP G) /\ ~(MULTI_EDGE G)"
```

HOL Definition 6 (FINITE_GRAPH_DEF)

```
"FINITE_GRAPH (G:~Graph) =
  (GRAPH G) /\ FINITE (VS G) /\ FINITE (ES G)"
```

Other abstract properties of graphs can be defined in a similar way.

4.1.3 Relationship between vertices and edges

Incidence An edge is said to be *incident with* the vertices which are the source or destination of the edge. It is said to be incident *from* the source vertex and to be incident *to* the destination vertex. The function **INCIDENT_WITH**, applied to a graph G and a vertex v , returns a set of edges which incident with the vertex v .

HOL Definition 7 (INCIDENT_WITH_DEF)

```
"INCIDENT_WITH (G:~Graph) v =
  {e | (e IS_EDGE G) /\ ((e_src e = v) \/ (e_des e = v))}"
```

Let us name the graph in Figure 4.1(a) as G' , then

INCIDENT_WITH G' 1

is equal to $\{(1,2,a),(4,1,d),(1,5,i)\}$. Similarly, we can define **INCIDENT_FROM** and **INCIDENT_TO** in HOL as below:

HOL Definition 8 (INCIDENT_FROM_DEF)

```
"INCIDENT_FROM (G:~Graph) v =
  {e | (e IS_EDGE G) /\ (e_src e = v) }"
```

HOL Definition 9 (INCIDENT_TO_DEF)

```
"INCIDENT_TO (G:"Graph) v =
  {e | (e IS_EDGE G) /\ (e_dsa e = v)}"
```

Degree The *degree* of a vertex is the total number of edges incident with it. The *out-degree* of a vertex is the number of edges incident from it and the *in-degree* is the number of edges incident to it. The HOL definition of degree make use of the definitions of incidence and the cardinal number of sets. They are listed below:

HOL Definition 10 (OUT_DEGREE_DEF)

```
"OUT_DEGREE (G:"Graph) v = CARD (INCIDENT_FROM G v)"
```

HOL Definition 11 (IN_DEGREE_DEF)

```
"IN_DEGREE (G:"Graph) v = CARD (INCIDENT_TO G v)"
```

HOL Definition 12 (DEGREE_DEF)

```
"DEGREE (G:"Graph) v =
  (IN_DEGREE G v) + (OUT_DEGREE G v)"
```

Thus, DEGREE G' 1 is 3, OUT_DEGREE G' 1 is 2 and IN_DEGREE G' 1 is 1.

Adjacence Two vertices are said to be *adjacent* if and only if there exists an edge between them. The predicate VER_ADJA $G v_1 v_2$ is true if there is an edge (v_1, v_2, x) or (v_2, v_1, y) for some x and y . The HOL definition of VER_ADJA is:

HOL Definition 13 (VER_ADJA_DEF)

```
"VER_ADJA G v1 (v2:e) =
  (GRAPH G) /\ (v1 IS_VERTEX G) /\ (v2 IS_VERTEX G) /\
  (?(e:"Edge). (e IS_EDGE G) /\
    (((e_src e = v1) /\ (e_dsa e = v2)) /\
     ((e_src e = v2) /\ (e_dsa e = v1))))"
```

In Figure 4.1(a), $\text{VER_ADJA } G' 12$ is T while $\text{VER_ADJA } G' 13$ is F. Similarly, two edges are adjacent if there is a vertex which is the destination of one and the source of the other.

HOL Definition 14 (E_ADJA_DEF)

```
"E_ADJA G e1 (e2:"Edge) =
  (GRAPH G) /\ (e1 IS_EDGE G) /\ (e2 IS_EDGE G) /\
  ((e_des e1 = e_src e2) /\ (e_des e2 = e_src e1))"
```

Successor and predecessor A vertex v_2 is a *successor* of another vertex v_1 if and only if there exists an edge from v_1 to v_2 . The predicate IS_SUC_VER defined below indicates this relationship.

HOL Definition 15 (IS_SUC_VER_DEF)

```
"IS_SUC_VER (G:"Graph) v1 v2 =
  ?e. (e IS_EDGE G) /\ (e_src e = v1) /\ (e_des e = v2)"
```

The vertex 2 in G' is the successor of vertex 1. The converse of successor is the *predecessor*. The corresponding predicate is IS_PRE_VER :

HOL Definition 16 (IS_PRE_VER_DEF)

```
"IS_PRE_VER (G:"Graph) v1 v2 =
  ?e. (e IS_EDGE G) /\ (e_des e = v1) /\ (e_src e = v2)"
```

The functions SUC_VERS and PRE_VERS return the set of vertices which are successors and predecessors, respectively.

HOL Definition 17 (SUC_VERS_DEF)

```
"SUC_VERS (G:"Graph) v =
  {v' | (v' IS_VERTEX G) /\ (IS_SUC_VER G v v')}"
```

HOL Definition 18 (PRE_VERS_DEF)

```
"PRE_VERS (G:"Graph) v =
  {v' | (v' IS_VERTEX G) /\ (IS_PRE_VER G v v')}"
```

Referring to Figure 4.1(a), $SUC.VERS\ G'1 = \{2, 5\}$ and $PRE.VERS\ G'1 = \{4\}$.

4.1.4 Operations on graphs

Insertion and deletion The primitive operations on graphs are insertion and deletion of a vertex or an edge. The definition of inserting a vertex is:

HOL Definition 19 (INSERT_VERTEX.DEF)

```
"INSERT_VERTEX v (G:"Graph) = (v INSERT (VS G), (ES G))"
```

and the definition of inserting an edge is:

HOL Definition 20 (INSERT_EDGE.DEF)

```
"INSERT_EDGE e (G:"Graph) =
  ((VS G),
   (((e_src e) IS_VERTEX G) /\ ((e_dst e) IS_VERTEX G)) =>
    (e INSERT (ES G)) | (ES G)))"
```

Note that to maintain the integrity of a graph, the only edges which can be inserted are those incident with vertices already in the graph. The reverse operations of insertion is DELETE_VERTEX and DELETE_EDGE. Their definitions are listed below:

HOL Definition 21 (DELETE_VERTEX.DEF)

```
"DELETE_VERTEX (G:"Graph) v =
  (((VS G) DELETE v), ((ES G) DIFF (INCIDENT_WITH v)))"
```

HOL Definition 22 (DELETE_EDGE.DEF)

```
"DELETE_EDGE (G:"Graph) e =
  ((VS G), ((ES G) DELETE e))"
```

Note also that deleting a vertex must also delete all the edges incident with it. The following four theorems assert that the abstract property of a graph is maintained over these operations.

HOL Theorem 4 (GRAPH_INSERT_VERTEX)

$$\vdash \forall G v. \text{GRAPH } G \supset \text{GRAPH } (v \text{ INSERT_VERTEX } G)$$

HOL Theorem 5 (GRAPH_INSERT_EDGE)

$$\vdash \forall G e. \text{GRAPH } G \supset \text{GRAPH } (e \text{ INSERT_EDGE } G)$$

HOL Theorem 6 (GRAPH_DELETE_VERTEX)

$$\vdash \forall G v. \text{GRAPH } G \supset \text{GRAPH } (G \text{ DELETE_VERTEX } v)$$

HOL Theorem 7 (GRAPH_DELETE_EDGE)

$$\vdash \forall G e. \text{GRAPH } G \supset \text{GRAPH } (G \text{ DELETE_EDGE } e)$$

All of these operations are commutative. These facts are asserted by the following theorems:

HOL Theorem 8 (INSERT_VERTEX_COMM)

$$\vdash \forall G v_1 v_2.$$

$$v_1 \text{ INSERT_VERTEX } (v_2 \text{ INSERT_VERTEX } G) =$$

$$v_2 \text{ INSERT_VERTEX } (v_1 \text{ INSERT_VERTEX } G)$$

HOL Theorem 9 (INSERT_EDGE_COMM)

$$\vdash \forall G e_1 e_2.$$

$$e_1 \text{ INSERT_EDGE } (e_2 \text{ INSERT_EDGE } G) =$$

$$e_2 \text{ INSERT_EDGE } (e_1 \text{ INSERT_EDGE } G)$$

HOL Theorem 10 (DELETE_VERTEX_COMM)

$$\vdash \forall G v_1 v_2.$$

$$(G \text{ DELETE.VERTX } v_1) \text{ DELETE.VERTX } v_2 =$$

$$(G \text{ DELETE.VERTX } v_2) \text{ DELETE.VERTX } v_1$$

HOL Theorem 11 (DELETE_EDGE_COMM)

$$\vdash \forall G \, e_1 \, e_2.$$

$$(G \text{ DELETE.EDGE } e_1) \text{ DELETE.EDGE } e_2 =$$

$$(G \text{ DELETE.EDGE } e_2) \text{ DELETE.EDGE } e_1$$

Graph union and intersection Two important operations on graphs are the *union* and *intersection* of two graphs. The union of two graphs G_1 and G_2 is defined to be the unions of their vertex sets and edge sets. The HOL definition reads:

HOL Definition 25 (G_UNION_DEF)

```
"G_UNION (G1:~Graph) G2 =
  ((VS G1) UNION (VS G2), (ES G1) UNION (ES G2))"
```

The operation G.UNION is closed within the set of all graphs, i.e., the union of any two graphs is a graph.

HOL Theorem 12 (GRAPH_UNION)

$$\vdash \forall G_1 \, G_2.$$

$$\text{GRAPH } G_1 \wedge \text{GRAPH } G_2 \supset \text{GRAPH } (G_1 \text{ G.UNION } G_2)$$

This operation is symmetric, associative and the union of a graph with itself results in itself. These properties are asserted by the following three theorems.

HOL Theorem 13 (G_UNION_SYM)

$$\vdash \forall G_1 \, G_2. \, G_1 \text{ G.UNION } G_2 = G_2 \text{ G.UNION } G_1$$

HOL Theorem 14 (G.UNION_ASSOC)

$$\vdash \forall G_1 G_2 G_3.$$

$$(G_1 \text{ G.UNION } G_2) \text{ G.UNION } G_3 = G_1 \text{ G.UNION } (G_2 \text{ G.UNION } G_3)$$

HOL Theorem 15 (G.UNION_IDENT)

$$\vdash \forall G. G \text{ G.UNION } G = G$$

It is obvious that if v is a vertex of the union of G_1 and G_2 , then it is either a vertex of G_1 or a vertex of G_2 . Similarly, if e is an edge of the union of G_1 and G_2 , then it is either an edge of G_1 or an edge of G_2 . Hence, the following two theorems:

HOL Theorem 16 (VERTEX_IN_UNION)

$$\vdash \forall G_1 G_2 v.$$

$$v \text{ IS.VERTEX } (G_1 \text{ G.UNION } G_2) = v \text{ IS.VERTEX } G_1 \vee v \text{ IS.VERTEX } G_2$$

HOL Theorem 17 (EDGE_IN_UNION)

$$\vdash \forall G_1 G_2 e. e \text{ IS.EDGE } (G_1 \text{ G.UNION } G_2) = e \text{ IS.EDGE } G_1 \vee e \text{ IS.EDGE } G_2$$

The definition of graph intersection is the intersections of their vertex sets and edge sets:

HOL Definition 24 (G.INTER_DEF)

$$\text{"G.INTER (G1: Graph) G2 = } \\ \text{(((VS G1) INTER (VS G2)), ((ES G1) INTER (ES G2)))}"$$

This operation is closed within the set of all graphs. This is expressed in the following theorem:

HOL Theorem 18 (GRAPH_INTER)

$$\vdash \forall G_1 G_2. \text{GRAPH } G_1 \wedge \text{GRAPH } G_2 \supset \text{GRAPH } (G_1 \text{ G.INTER } G_2)$$

And this operation is also symmetric, associative and reflexive, hence, the following three theorems:

HOL Theorem 19 (G.INTER.SYM)

$$\vdash \forall G_1 G_2. G_1 \text{ G.INTER } G_2 = G_2 \text{ G.INTER } G_1$$

HOL Theorem 20 (G.INTER.ASSOC)

$$\vdash \forall G_1 G_2 G_3. (G_1 \text{ G.INTER } G_2) \text{ G.INTER } G_3 = G_1 \text{ G.INTER } (G_2 \text{ G.INTER } G_3)$$

HOL Theorem 21 (G.INTER.IDENT)

$$\vdash \forall G. G \text{ G.INTER } G = G$$

All vertices of the intersection of two graphs must be the vertices of both of the graphs. Similarly, all edges of the intersection must be the edges of these graphs.

The following two theorems assert these facts:

HOL Theorem 22 (VERTEX.IN.INTER)

$$\vdash \forall G_1 G_2 v. v \text{ IS.VERTEX } (G_1 \text{ G.INTER } G_2) = v \text{ IS.VERTEX } G_1 \wedge v \text{ IS.VERTEX } G_2$$

HOL Theorem 23 (EDGE.IN.INTER)

$$\vdash \forall G_1 G_2 e. e \text{ IS.EDGE } (G_1 \text{ G.INTER } G_2) = e \text{ IS.EDGE } G_1 \wedge e \text{ IS.EDGE } G_2$$

4.1.5 Subgraphs and graph isomorphism

Subgraph A subgraph of a graph G is a graph whose vertex set and edge set are subsets of the vertex set and edge set of G , respectively. A predicate **SUBGRAPH** is defined for this relation.

HOL Definition 25 (SUBGRAPH_DEF)

"SUBGRAPH (H: 'Graph) (G: 'Graph) =
 (GRAPH H) \wedge (GRAPH G) \wedge
 ((\forall E) SUBSET (VS G)) \wedge ((\exists E) SUBSET (ES G))"

SUBGRAPH $H\ G$ is true if H is a subgraph of G . As the definition implies, a subgraph is itself a graph. This is asserted by the theorem SUBGRAPH_GRAPH.

HOL Theorem 24 (SUBGRAPH_GRAPH)

$$\vdash \forall G\ H. \text{SUBGRAPH } H\ G \supset \text{GRAPH } G \wedge \text{GRAPH } H$$

The subgraph relation is reflexive, transitive and antisymmetric. These properties are asserted by the following three theorems.

HOL Theorem 25 (SUBGRAPH_REFL)

$$\vdash \forall G. \text{GRAPH } G \supset \text{SUBGRAPH } G\ G$$

HOL Theorem 26 (SUBGRAPH_TRANS)

$$\vdash \forall G_1\ G_2\ G_3. \text{SUBGRAPH } G_1\ G_2 \wedge \text{SUBGRAPH } G_2\ G_3 \supset \text{SUBGRAPH } G_1\ G_3$$

HOL Theorem 27 (SUBGRAPH_ANTISYM)

$$\vdash \forall G_1\ G_2. \text{SUBGRAPH } G_1\ G_2 \wedge \text{SUBGRAPH } G_2\ G_1 \supset (G_1 = G_2)$$

From the definition, a subgraph can be obtained by deleting an edge and/or a vertex from a graph. This is expressed in the following two theorems:

HOL Theorem 28 (SUBGRAPH_DELETE_EDGE)

$$\vdash \forall G\ e. \text{GRAPH } G \supset \text{SUBGRAPH } (G\ \text{DELETE_EDGE } e)\ G$$

HOL Theorem 29 (SUBGRAPH_DELETE_VERTEX)

$$\vdash \forall G v. \text{GRAPH } G \supset \text{SUBGRAPH } (G \text{ DELETE_VERTEX } v) G$$

A subgraph can also be obtained by applying selection functions to the vertex set and the edge set of a graph. This operation is defined in HOL as:

HOL Definition 26 (MK_SUBGRAPH_DEF)

```
"MK_SUBGRAPH (G:"Graph) fv fe =
  {v | v IS_VERTEX G /\ fv v},
  {a | a IS_EDGE G /\ fa a /\ fv (a_src a) /\ fv (a_dst a)}"
```

The theorem MK_SUBGRAPH_GRAPH asserts that the result of this operation maintains the integrity of graph, and the theorem MK_SUBGRAPH_SUBGRAPH asserts that the result is indeed a subgraph of the original graph.

HOL Theorem 30 (MK_SUBGRAPH_GRAPH)

$$\vdash \forall G fv fe. \text{GRAPH } G \supset \text{GRAPH } (\text{MK_SUBGRAPH } G fv fe)$$

HOL Theorem 31 (MK_SUBGRAPH_SUBGRAPH)

$$\vdash \forall G fv fe. \text{GRAPH } G \supset \text{SUBGRAPH } (\text{MK_SUBGRAPH } G fv fe) G$$

Graph isomorphism Two graphs G_1 and G_2 are isomorphic if there exists a one-one correspondence between the vertices and edges of G_1 and the vertices and edges of G_2 , respectively. The predicate GRAPH_ISO is defined for this relation:

HOL Definition 27 (GRAPH_ISO_DEF)

```
"GRAPH_ISO (G:"Graph) (H:"Graph) (f,g) =
  (GRAPH G) /\ (GRAPH H) /\
  ((\v G) <--> (\v H)) /\ ((\e G) <--> (\e H))g"
```

where the infix constant \longleftrightarrow means one-one correspondence. In the above definition, $((\forall S \ G) \longleftrightarrow (\forall S \ H)) \ f$ means f is a one-one correspondence between the elements of the vertex set of G and the elements of the vertex set of H .

Four theorems about graph isomorphism have been proved. They assert the properties of this relation, namely reflexive (automorphism), transitive and symmetric.

HOL Theorem 32 (GRAPH_ISO_AUTO)

$$\vdash \forall G. \text{GRAPH } G \supset \text{GRAPH_ISO } G \ G \ (I, I)$$

HOL Theorem 33 (GRAPH_ISO_TRANS)

$$\begin{aligned} &\vdash \forall g_1 \ g_2 \ g_3 \ f_1 \ g_1 \ f_2 \ g_2. \\ &\quad \text{GRAPH_ISO } g_1 \ g_2 \ (f_1, g_1) \wedge \text{GRAPH_ISO } g_2 \ g_3 \ (f_2, g_2) \supset \\ &\quad \text{GRAPH_ISO } g_1 \ g_3 \ ((f_2 \circ f_1), (g_2 \circ g_1)) \end{aligned}$$

HOL Theorem 34 (GRAPH_ISO_SYM)

$$\vdash \forall G \ H \ f \ g. \text{GRAPH_ISO } G \ H \ (f, g) \supset (\exists f' \ g'. \text{GRAPH_ISO } H \ G \ (f', g'))$$

HOL Theorem 35 (GRAPH_ISO_SYM_INV)

$$\begin{aligned} &\vdash \forall G \ H \ f \ g. \text{GRAPH_ISO } G \ H \ (f, g) \supset \\ &\quad \text{GRAPH_ISO } H \ G \ (\text{FUN_INV } (\forall S \ G) (\forall S \ H) \ f, \text{FUN_INV } (\text{ES } G) (\text{ES } H) \ g) \end{aligned}$$

The theorem **GRAPH_ISO_SYM_INV** makes a stronger assertion about the symmetry of graph isomorphism by explicitly providing an inverse function **FUN_INV** where the expression **FUN_INV** $S_1 \ S_2 \ f$ is an inverse function of f , and its domain is the set S_2 and its range is the set S_1 .

4.2 The theory path

One of the most important uses of graphs with respect to the applications in railway signalling systems is the derivation of paths. The path theory contains definitions of a path and related constants. Some basic properties of paths have been proved.

Consider any two vertices v_1 and v_2 in a graph, v_2 is *reachable* from v_1 if there is a sequence of edges through which one can arrive at v_2 from v_1 . There are usually many different ways one can arrive at v_2 . According to whether all the edges in the sequence are distinct, the sequences can be classified into several classes. They are *walks*, *trails* and *paths*.

4.2.1 Walks in a graph

A *walk* in a graph G is a sequence of edges e_1, e_2, \dots, e_n , which satisfies the following:

1. $n > 0$;
2. e_i is an edge of G , for all $i = 1, \dots, n$;
3. the destination of e_i is equal to the source of e_{i+1} for $1 \leq i < n$.

This implies that the edges are not necessarily distinct in a walk, i.e., a walk may pass through the same edge more than once. In HOL, a sequence of edges is represented by a list of edges, of type $(\text{'Edge'})\text{list}$. A list of edges satisfies the predicate **WALK** if and only if it is a walk.

HOL Definition 28 (WALK_DEF)

```
"WALK g (w:('Edge')list) =
  ~(NULL w) /\ (WALK_TAIL w g)"
```

HOL Definition 29 (WALK_TAIL_DEF)

```
"(WALK_TAIL [] (G:"Graph) = T) /\
  ((hd:"Edge) t1. WALK_TAIL (CONS hd t1) G =
    (GRAPH G) /\ (hd IS_EDGE G)) /\
    ((NULL t1) \\/ ((WALK_TAIL t1 G) /\ (a_des hd = a_src (HD t1))))"
```

Here, the recursive predicate **WALK_TAIL** guarantees the list of edges forms a walk by checking whether the conditions 2 and 3 listed above are met. The degenerate case, the null list, is not to be considered as a walk. The *entry* of a walk is the source of the first edge in the list, and the *exit* of a walk is the destination of the last edge.

HOL Definition 30 (WALK_ENTRY_DEF)

```
"WALK_ENTRY (l:("Edge)list) = a_src (HD l)"
```

HOL Definition 31 (WALK_EXIT_DEF)

```
"WALK_EXIT (CONS (hd:"Edge) t1) =
  (NULL t1 => (a_des hd) | (WALK_EXIT t1))"
```

4.2.2 Some operations and facts on sequences of edges

Since the lists representing walks and other classes of edge sequences are special cases of general lists, the operations, functions and theorems in the HOL system theory **list** are not sufficient. Several predicates are defined to deal with these edge sequences, and are described in this subsection, together with some theorems about them.

Membership of a list The concept of membership is borrowed from set theory.

An object x is a member of a list $[x_0, \dots, x_n]$ if $x = x_i$ for some i where $0 \leq i \leq n$.

The predicate **ELEM** is defined to have this list membership property.

HOL Definition 32 (ELEM_DEF)

$$\begin{aligned} & \text{"(ELEM } \sqcap (x:o) = F) \wedge \\ & \text{(ELEM (CONS } h \ t) (x:o) = (x = h) \vee (ELEM \ t \ x))\text{"} \end{aligned}$$

It is obvious that there is no element in a null list. If x is in a list l , it is also in the list obtained by adding an element to l . If x is an element of the list obtained by appending a list l_2 to a list l_1 , it is an element of l_1 or l_2 . These facts are asserted by the theorems **NULL_NOT_ELEM**, **ELEM_CONS** and **ELEM_APPEND**.

HOL Theorem 36 (NULL_NOT_ELEM)

$$\vdash \forall l. \text{NULL } l \supset \forall x. \neg(\text{ELEM } l \ x)$$
HOL Theorem 37 (ELEM_CONS)

$$\vdash \forall x \ y. \text{ELEM } l \ x \supset \text{ELEM (CONS } y \ l) \ x$$
HOL Theorem 38 (ELEM_APPEND)

$$\vdash \forall l_1 \ l_2 \ x. \text{ELEM (APPEND } l_1 \ l_2) \ x = \text{ELEM } l_1 \ x \vee \text{ELEM } l_2 \ x$$

The theorem **IN_ELEM** relates the set membership with the list membership. It asserts that there exists a list l such that all members of a finite set s are elements of l .

HOL Theorem 39 (IN_ELEM)

$$\vdash \forall s. \text{FINITE } s \supset (\exists l. (\forall x. x \text{ IN } s = \text{ELEM } l \ x))$$

Unique elements In a list $[x_0, \dots, x_n]$, the elements are called *unique elements* if all x are distinct. The recursive function **UNIQUE_EL** is defined to check the uniqueness of the elements of a list, and is used in the definitions of **trail** and **path**.

HOL Definition 33 (UNIQUE_EL_DEF)

```
"(UNIQUE_EL [] = T) /\
  (UNIQUE_EL (CONS (hd:*)t1) = (EVERY (\x. ~(x = hd)) t1) /\ (UNIQUE_EL t1))"
```

Element set A set can be constructed to contain all elements of a list. Obviously, such a set will contain all distinct elements of the list. This provides a means of collecting all distinct elements of a list, and applying set operations on them, for example, checking whether two lists have common elements can be performed by using the set disjoint predicate on the element sets (a definition corresponding to this is shown in **HOL Definition 35**). The function **EL_SET** returns a set containing all element of its argument list.

HOL Definition 34 (EL_SET_DEF)

```
"(EL_SET [] = {}) /\
  (EL_SET (CONS hd t1: list) = hd INSERT (EL_SET t1))"
```

The element set of the list (**APPEND** l_1 l_2) is the union of the element sets of l_1 and l_2 .

HOL Theorem 40 (EL_SET_APPEND)

$$\vdash \forall l_1 l_2. \text{EL_SET} (\text{APPEND } l_1 l_2) = \text{EL_SET } l_1 \text{ UNION } \text{EL_SET } l_2$$

The theorem **ELEM_IN_EL_SET** asserts that the list membership relation is equivalent to the set membership of the element set.

HOL Theorem 41 (ELEM_IN_EL_SET)

$$\vdash \forall x. \text{ELEM } i x = x \text{ IN } \text{EL_SET } i$$

Disjoint lists Two lists are said to be *disjoint* if they do not have common elements. The predicate `DISJ_LIST` is defined for testing this condition. It is defined in the way suggested above.

HOL Definition 35 (`DISJ_LIST_DEF`)

"`DISJ_LIST (l1:(*)list) l2 = DISJOINT (EL_SET l1) (EL_SET l2)`"

The basic properties of `DISJ_LIST` follow those of the set operator `DISJOINT`, i.e., it is symmetric.

HOL Theorem 42 (`DISJ_LIST_COMM`)

$$\vdash \forall l_1 l_2. \text{DISJ_LIST } l_1 l_2 = \text{DISJ_LIST } l_2 l_1$$

The following two theorems state the facts about `DISJ_LIST` over the list operators `CONS` and `APPEND`.

HOL Theorem 43 (`DISJ_LIST_CONS`)

$$\vdash \forall l_1 l_2 h. \text{DISJ_LIST } (\text{CONS } h l_1) l_2 = \text{DISJ_LIST } l_1 l_2 \wedge \neg \text{ELEM } l_2 h$$

HOL Theorem 44 (`DISJ_LIST_APPEND`)

$$\vdash \forall l_1 l_2 l_3. \text{DISJ_LIST } (\text{APPEND } l_1 l_2) l_3 = \text{DISJ_LIST } l_1 l_3 \wedge \text{DISJ_LIST } l_2 l_3$$

Vertex lists The functions described in the remaining of this subsection are used in the reasoning of paths. They are meaningful only when applying to the edge sequences which are walks in a graph.

Consider a walk $w = [e_0; \dots; e_n]$ in a graph, the vertices passed through by w are the source of e_0 and the destinations of e_i for $0 \leq i \leq n$. The function `VER_LIST` returns the list of vertices a walk passes through. It is defined in terms of `V_L` which returns the same list except the entry vertex.

HOL Definition 36 (VER_LIST_DEF)

```
"(VER_LIST [] = []) /\
  (VER_LIST (CONS hd:~Edge) tl) = CONS (e_arc hd) (V_L (CONS hd tl)))"
```

HOL Definition 37 (V_L_DEF)

```
"(V_L [] = []) /\
  (V_L (CONS (hd:~Edge) tl) = CONS (e_des hd) (V_L tl))"
```

The following three theorems state the properties of the function VER_LIST over the list operators CONS and APPEND.

HOL Theorem 45 (V_L_APPEND)

$$\vdash \forall p_1 p_2. V_L (APPEND p_1 p_2) = APPEND (V_L p_1) (V_L p_2)$$
HOL Theorem 46 (VER_LIST_CONS)

$$\vdash \forall p h. VER_LIST (CONS h p) = CONS (e_arc h) (CONS (e_des h) (V_L p))$$
HOL Theorem 47 (VER_LIST_APPEND)

$$\vdash \forall p_1 p_2. \neg NULL p_1 \wedge \neg NULL p_2 \supset \\ (VER_LIST (APPEND p_1 p_2) = APPEND (VER_LIST p_1) (TL (VER_LIST p_2)))$$
4.2.3 Trails and paths

A *trail* is a walk which contains no repeated edges, i.e., all edges in the sequence are distinct. However, it may pass through the same vertex more than once, thus containing a cycle.

HOL Definition 38 (TRAIL_DEF)

```
"TRAIL (g:~Graph) (l:~Edge)list =
  (WALK g l) /\ (UNIQUE_EL l)"
```

The clause `UNIQUE_EL l` makes sure that all elements in the list *l* are distinct, i.e., no edges in a trail are repeated.

A *path* is a trail which passes through any vertex at most once, i.e., there is no cycle in a path. `PATH G l` if and only if *l* is a path in the graph *G*.

HOL Definition 39 (`PATH_DEF`)

```
"PATH (G:"Graph) (l:("Edge)list) =
  (TRAIL G l) /\ (UNIQUE_EL (VER_LIST l))"
```

The clause `UNIQUE_EL (VER_LIST l)` guarantees that all vertices passed through by *l* are distinct. In the application in railway signalling, paths are the most important type of lists, therefore, some theorems about paths will be described in the next subsection.

The entry of a path is defined to be the source vertex of the first edge in the sequence:

HOL Definition 40 (`PATH_ENTRY_DEF`)

```
"PATH_ENTRY (l:("Edge)list) = e_src (ED 1)"
```

and the exit of a path is the destination vertex of the last edge in the sequence. It is defined in terms of the exit of a walk (`WALK_EXIT`).

HOL Definition 41 (`PATH_EXIT_DEF`)

```
"PATH_EXIT (p:("Edge)list) = WALK_EXIT p"
```

4.2.4 Some properties of paths

Disjoint paths Two paths p_1 and p_2 are said to be *disjoint* if they do not overlap, i.e., they do not share any edges, nor have identical vertices. A HOL definition for this may be

$DISJ_LIST\ p1\ p2 \wedge DISJ_LIST\ (VER_LIST\ p1)\ (VER_LIST\ p2)$

The actual definition in the theory is $DISJ_PATH_DEF$ which replaces the constants VER_LIST by $V.L.$, thus, it excludes the entry vertices in the disjoint test. This is needed to overcome a difficulty when $DISJ_PATH$ is used to test two paths which are to be connected to form a longer path. In such case, the exit of one path should be equal to the entry of the other.

HOL Definition 42 ($DISJ_PATH_DEF$)

$\neg DISJ_PATH\ G\ p1\ p2 = PATH\ G\ p1 \wedge PATH\ G\ p2 \wedge$
 $DISJ_LIST\ p1\ p2 \wedge DISJ_LIST\ (V.L\ p1)\ (V.L\ p2)$

Extending a path An existing path p can be extended by adding an edge h to the front of it. The theorem $PATH_CONS$ expresses the conditions that a path can be extended in this way. The conditions are:

1. h must be an edge in the same graph;
2. the entry of p is equal to the destination of h ;
3. h is not already an element of p ;
4. the source of h is not equal to any of the vertices in p .

HOL Theorem 48 ($PATH_CONS$)

$\vdash \forall p \wedge G.$

$GRAPH\ G \wedge PATH\ G\ p \wedge h\ IS_EDGE\ G \wedge (PATH_ENTRY\ p = e_des\ h) \wedge$

$\neg LOOP\ h \wedge \neg ELEM\ (VER_LIST\ p)\ (e_src\ h) \wedge \neg ELEM\ p\ h \supset$

$PATH\ G\ (CONS\ h\ p)$

Two existing paths p_1 and p_2 can also be concatenated using the list operator **APPEND** to form a new path whose entry is the entry of p_1 and whose exit is the exit of p_2 , providing the following conditions hold:

1. the exit of p_1 is equal to the entry of p_2 ;
2. p_1 and p_2 are disjoint paths, i.e., $\text{DISJ.PATH } G \ p_1 \ p_2$;
3. the entry of p_1 is not equal to any of the vertices in p_2 ;

The first condition guarantees that the resulting path is connected, and the second and third conditions eliminate the possibility that the resulting path will have repeated edges and/or loops. This is expressed in the theorem **PATH.APPEND**.

HOL Theorem 49 (PATH.APPEND)

$\vdash \forall G \ p_1 \ p_2.$

$\text{GRAPH } G \wedge \text{DISJ.PATH } G \ p_1 \ p_2 \wedge (\text{PATH.EXIT } p_1 = \text{PATH.ENTRY } p_2) \wedge$

$\neg \text{ELEM}(\text{VER.LIST } p_2)(\text{PATH.ENTRY } p_1) \supset$

$\text{PATH } G(\text{APPEND } p_1 \ p_2)$

Paths under graph operations If p is a path in G_1 , then it is still a path in the union of G_1 and another graph, say G_2 .

HOL Theorem 50 (PATH.G.UNION)

$\vdash \forall p \ G_1 \ G_2.$

$\text{GRAPH } G_1 \wedge \text{GRAPH } G_2 \wedge \text{PATH } G_1 \ p \supset \text{PATH}(G_1 \ \text{G.UNION } G_2) \ p$

If p is a path in G , then it is a path of the graph resulting from inserting an edge or a vertex into G .

HOL Theorem 51 (PATH_INS_EDGE)

$$\vdash \forall p \in G. \text{PATH } G \ p \supset \text{PATH } (e \text{ INSERT_EDGE } G) \ p$$

HOL Theorem 52 (PATH_INS_VERTEX)

$$\vdash \forall p \in G. \text{PATH } G \ p \supset \text{PATH } (v \text{ INSERT_VERTEX } G) \ p$$

Connected graph Finally, the concept of *connected graph* is defined in terms of whether there is a path connecting any two vertices in the graph.

HOL Definition 43 (CONNECTED_DEF)

```
"CONNECTED G = GRAPH G /\
  ((v1 v2. (v1 IS_VERTEX G) /\ (v2 IS_VERTEX G) /\ ~(v1 = v2)
  ==>
  (v1. (PATH G 1) /\ (v1 = PATH_ENTRY 1) /\ (v2 = PATH_EXIT 1)))"
```

The theories described in this chapter form a mathematical foundation for modelling the track network. These theories have been developed in a very general way to anticipate the needs of other applications. They have been organized into a library which can be loaded into the HOL system by a simple command. When other applications call for the use of graph, this library will be a quick and reasonable starting point.

Chapter 5

Modelling of Railway

Components

This chapter describes the theories which model the individual track components and signals.

Let us now consider how to model the railway track components and signals. Their basic functions have been described in Chapter 3. The key to the modelling is abstraction. The basic principle in the development of the theories modelling these components is to concentrate on the major function of each of them. An abstract type is defined to represent each class of components. The basic functions of the components are encoded in the properties of these types. The types are defined using the type definition package[48] in the HOL system described briefly on Page 24. Then, appropriate projection operators and discriminators are defined to manipulate objects of these types. These types and constant definitions and theorems about their basic properties are arranged in three HOL theories: **TRACK**, **SIGNAL** and **PART**. Each of these theories is described in detail in a separate section

below.

5.1 The theory TRACK

This theory contains type definitions and constant definitions about the individual track components, namely joins, track circuits and points. The complete theory is listed in Appendix A.6 and the ML source creating this theory is listed in Appendix B.7.

5.1.1 Joins

Since joins have no moving parts, all that is required to characterize a join is its type. Simply, an enumerated type is defined to represent them. There are three types of joins in the real track network as listed in Section 3.1. In addition, a special type of join is required to indicate the connection point between the two areas under different control centres. Therefore, the enumerated type has four possible values.

The name of the type is `Join`, its specification is

HOL Definition 44 (Join.Axiom)

```
'Join = J_conduct | J_insulate | J_overlap | J_terminate'
```

The value `J_terminate` is for the joins between control areas. Four predicates, `IS_JCOND`, `IS_JINSU`, `IS_JOVER` and `IS_JTERM` are defined for testing the value of a join. They return `T` if they are applied to a join whose value is `J_conduct`, `J_insulate`, `J_overlap` or `J_terminate`, respectively.

5.1.2 Track circuits

At any given time, a track circuit is in one of the two physical states, either 'CLEAR' or 'OCCUPIED'. When a route has been set up, it locks the sections of track so that a conflicting route cannot be set up. Although the track circuits along the route are not in the 'OCCUPIED' state, they cannot be included into another route. They are said to be in a 'LOCKED' state. The track circuit state is represented by an enumerated type :Tstate with the following specification:

HOL Definition 45 (Tstate_Axiom)

'Tstate = occupied | locked | clear'

The three constant values correspond to the physical states 'CLEAR', 'OCCUPIED', and the logical state 'LOCKED'.

A track circuit is represented by the type :Tcix with the following specification:

HOL Definition 46 (Tcix_Axiom)

'Tcix = TCIR num (num->Tstate)'

The first field of a track circuit is its identification number, which is of type :num. The second field is a function of time yielding the current state of the circuit. For example, if a track circuit is occupied at the time slot 10, then $S_{tc10} = \text{occupied}$, where S_{tc} is its state function. This time function represents the physical input into the system.

Within the abstract model of railway, time is represented by natural numbers, i.e., of type :num, thus, time is on a discrete scale. This is reasonable approximation of the real system providing the unit of time is sufficiently small. The actual unit depends on how the control system is implemented. In the case of SSI or similar

implementations, the time unit could be the duration of a control cycle. The origin of the time scale could be any fixed time in the past.

There are projection operators defined for accessing the fields of track circuits. They are `TC_ID` which returns the identification number, and `TC_SFUNC` which returns the state function.

5.1.3 Points

The modelling of points follows the same general principle of modelling track circuits. There are two sets of orthogonal states: one concerns with the physical states, the position of the point which can be either 'NORMAL', 'REVERSE' or moving between these static positions; the other set concerns with the logical states which indicate whether the point can be moved. The set of physical states is represented by the type `Ppos`. Its specification is

HOL Definition 47 (`Ppos.Axiom`)

`'Ppos = normal | reverse | moving'`

The set of logical state is represented by the type `Ploc` with the following specification:

HOL Definition 48 (`Ploc.Axiom`)

`'Ploc = free_move | free_nor_rev | free_rev_nor | remote_locked'`

where `free_move` indicates the point is free to move to any position, `free_nor_rev` indicates it is free to move from NORMAL to REVERSE, `free_rev_nor` indicates it is free to move from REVERSE to NORMAL, and `remoteLocked` indicates it cannot be moved at all.

A point is represented by the type `:Point` which contains three fields as shown in the specification below:

HOL Definition 49 (Point.Axiom)

```
'Point = POINT num (num->Ppos) (num->Ploc)'
```

The first field is the identification number, the second is the physical state function and the last the logical state function.

There are three projection operators corresponding to these fields, namely `PNT_ID`, `PNT_POS` and `PNT_LOC`. As it is often required to test the position of points, the predicates `PNT.NORMAL` and `PNT.REVERSE` are defined to yield `T` if the point is at the respective position.

HOL Definition 50 (PNT.NORMAL.DEF)

```
"PNT.NORMAL p t = ((PNT_POS p t) = normal)"
```

HOL Definition 51 (PNT.REVERSE.DEF)

```
"PNT.REVERSE p t = ((PNT_POS p t) = reverse)"
```

5.2 The theory SIGNAL

It has been mentioned in Chapter 3 that there are a number of classes of signals and that several signals from different classes may be combined on a signal post to form a compound signal. Following the basic principle, a type is defined to represent each class of signals. Another type based on these types of simple signals is defined to represent compound signals. The complete theory is listed in Appendix A.5 and the ML source creating this theory is listed in Appendix B.6.

5.2.1 Simple signals

Main signal Main signal is the most complex of all classes of simple signals because they can display up to 4-different aspects, and because there are different types according to the number of aspects can be displayed. The enumerated type :MAspect is defined for the current state of main signals.

HOL Definition 52 (MAspect.Axiom)

```
'MAspect = green | double_yellow | yellow | red
          | green_flash | double_yellow_flash | yellow_flash
          | faulty_aspect'
```

faulty_aspect indicates that the signal is faulty. All other values indicate that the chosen aspect is proved to be alright. Another enumerated type, namely Mtype is defined for distinguishing the kind of main signals, i.e., the number of aspects it can display.

HOL Definition 53 (Mtype.Axiom)

```
'Mtype = two_aspect | three_aspect | four_aspect
          | two_repeat | three_repeat'
```

The type for main signal is Msig which has two fields: the first indicates the kind of signal and the second is the state function.

HOL Definition 54 (Msig.Axiom)

```
'Msig = MSIG Mtype (num->MAspect)'
```

There are three predicates for testing the current state of a main signal, namely MAIN_ON, MAIN_OFF and MAIN_FAULTY. A main signal is said to be "ON" if the RED aspect is alright. It is said to be faulty if the state function returns the value faulty_aspect. Otherwise it is "OFF".

HOL Definition 55 (MAIN_ON_DEF)

"MAIN_ON s (t:num) = (N_ASPECT s t) = red"

HOL Definition 56 (MAIN_FAULTY_DEF)

"MAIN_FAULTY s (t:num) = (N_ASPECT s t) = faulty_aspect"

HOL Definition 57 (MAIN_OFF_DEF)

"MAIN_OFF s (t:num) = ~(MAIN_ON s t) /\ ~(MAIN_FAULTY s t)"

Junction indicators The type *Jsig* is defined for both junction indicators and route indicators regardless how they are implemented. The only thing which concerns us is whether the indicator is alight. A state function of type *num*→*bool* is used for the current state, where *T* (true) means the chosen arm or route number of the indicator is proved alight.

HOL Definition 58 (Jsig_Axiom)

'Jsig = JSIG (num→bool)'

Subsidiary signals A subsidiary signal has only the 'OFF' aspect which gives authority to the driver to pass the main signal showing the 'ON' aspect but prepare to stop short of any obstruction. Therefore, the type for subsidiary signal aspect has two possible values: *sub_not_show* and *sub_off*. The type representing subsidiary signals is *Subsig* which has only a state function returns the current aspect.

HOL Definition 59 (SubAspect_Axiom)

'SubAspect = sub_not_show | sub_off'

HOL Definition 60 (Subsig_Axiom)

'Subsig = SUBSIG (num→SubAspect)'

Shunting signals Shunting signals have two possible aspects: 'ON' and 'OFF', and may have a proving circuit for the 'ON' aspect; thus the type :ShAspect has three possible values. The type for shunting signal :Shsig has only one field, the state function.

HOL Definition 61 (ShAspect.Axiom)

```
'ShAspect = sh_on | sh_off | sh_faulty'
```

HOL Definition 62 (Shsig.Axiom)

```
'Shsig = SHUNTSIG (num->ShAspect)'
```

5.2.2 Compound signals

Compound signals are represented by the type :Signal. A constructor is provided for each combination of types of signals.

HOL Definition 63 (Signal.Axiom)

```
'Signal = SIGNALN num Nsig |
  SIGNALNJ num Nsig Jsig |
  SIGNALNS num Nsig Subsig |
  SIGNALNSJ num Nsig Subsig Jsig |
  SIGNALS num Shsig'
```

The first field of any compound signal is the identification number. The other fields are the constituent signals. The projection operators SIGNAL_ID and SIGNAL_MAIN are defined to access the identification number and the main signal.

HOL Definition 64 (SIGNAL_ID_DEF)

```
“(SIGNAL_ID (SIGNALN id n) = id) /\
 (SIGNAL_ID (SIGNALNJ id n j) = id) /\
 (SIGNAL_ID (SIGNALNS id n s) = id) /\
 (SIGNAL_ID (SIGNALNSJ id n s j) = id) /\
 (SIGNAL_ID (SIGNALS id sh) = id)”
```

HOL Definition 65 (SIGNAL.MAIN_DEF)

```
"(SIGNAL_MAIN (SIGNALM id m) = m) /\
(SIGNAL_MAIN (SIGNALMJ id m j) = m) /\
(SIGNAL_MAIN (SIGNALMS id m s) = m) /\
(SIGNAL_MAIN (SIGNALMSJ id m s j) = m) "
```

Since the 'ON' and 'OFF' states are of most importance in the operation of interlocking, two predicates, ON and OFF are defined for testing the current ON/OFF state of a signal.

HOL Definition 66 (ON_DEF)

```
"(ON (SIGNALM id m) t = (MAIN_ON m t)) /\
(ON (SIGNALMJ id m j) t = (MAIN_ON m t)) /\
(ON (SIGNALMS id m s) t = (MAIN_ON m t)) /\
(ON (SIGNALMSJ id m s j) t = (MAIN_ON m t)) /\
(ON (SIGNALS id sh) t = (SHUNT_ON sh t)) "
```

HOL Definition 67 (OFF_DEF)

```
"(OFF (SIGNALM id m) t = (MAIN_OFF m t)) /\
(OFF (SIGNALMJ id m j) t = (MAIN_OFF m t)) /\
(OFF (SIGNALMS id m s) t = (MAIN_OFF m t)) /\
(OFF (SIGNALMSJ id m s j) t = (MAIN_OFF m t)) /\
(OFF (SIGNALS id sh) t = (SHUNT_OFF sh t)) "
```

If a signal is neither 'ON' nor 'OFF', then it is faulty. the predicate SIGNAL_FAULT indicates such a state.

HOL Definition 68 (SIGNAL_FAULT_DEF)

```
"SIGNAL_FAULT = t = ~( (ON s t) /\ (OFF s t)) "
```

At any given time, a signal will be in either 'ON' or 'OFF' or 'FAULTY' state, and it will never be in both 'ON' and 'OFF' states characterized by the predicates ON, OFF and SIGNAL_FAULT, and they are in turn based on the properties that the constructors for the types of signal aspects are distinct and one-one. This is a

important property of the signals and it is asserted by the theorems `SIGNAL_STATES` and `SIGNAL_NOT_ON_OFF`.

HOL Theorem 53 (`SIGNAL_STATES`)

$$\vdash \forall st. (ON\ st) \vee (OFF\ st) \vee (SIGNAL_FAULT\ st)$$

HOL Theorem 54 (`SIGNAL_NOT_ON_OFF`)

$$\vdash \forall st. \neg((ON\ st) \wedge (OFF\ st))$$

5.3 The theory PART

This theory contains two type definitions defining two kinds of atomic building blocks for creating track networks. The first is the parts which will become the vertices in the graph representing the network, and the second is the labels of the edges. Chapter 6 will show how a network is formed using these parts and edges. Meanwhile, the definitions of parts and edge labels are described. The complete theory listing can be found in Appendix A.7 and the ML source creating this theory is listed in Appendix B.8.

5.3.1 Parts

The type `:Part` is defined to represent a section of track in the network. Each part has an identification number, an associated track circuit (except buffers) and a single atomic track component which may be any class of components listed in Table 3.1. A track circuit may be shared by more than one part.

HOL Definition 69 (`Part_Axiom`)

```
'Part = SPART num {
```

```
TPART num Tc1r |  
DPART num Tc1r (num#num) (num#num) |  
PPART num Tc1r Point (num#num#num)'
```

Since there are four kinds of atomic components, the definition of type :Part has four cases. A part constructed by BPART represents a buffer, and by TPART represents a section of plain track. They are simple and do not deserve more explanation.

A diamond crossing is represented by a DPART part. The last two fields, of type :num#num are the identification numbers of the adjacent parts. A movement through the diamond crossing can only be made between the parts indicated in the same pair.

Since all three kind of parts mentioned above contains no moving elements, they are static. In contrast, a PPART represents a section containing a point which may change state according to its position. The current state of a point is returned by the state function in the third field. The last field, a triple of identification numbers, pointing to the adjacent parts which should on the trailing, normal and reverse ends of the point, respectively.

Projection operators are defined for accessing the various fields of a part, and discriminator are defined for testing what kind of part an object of this type is. They are listed in Table 5.1.

5.3.2 Edge labels

The adjacent parts are connected by edges which is labelled by the join between the parts, and possibly a signal. The type :E1b1 is defined to represent the edge label. It has two cases: either a join with attached signal or simply a join.

PART.ID	returns the identification number of a part
PART.CIRCUIT	returns the track circuit associated with a part
PART.POINT	returns the point of a part
PART.PNT.TRAILING	returns the ID number of the part at the trailing end
PART.PNT.NORMAL	returns the ID number of the part at the normal end
PART.PNT.REVERSE	returns the ID number of the part at the reverse end
PART.DIA1	returns the first pair of ID numbers of the adjacent part
PART.DIA2	returns the second pair of ID numbers of the adjacent part
IS.BPART	T if the part is a buffer
IS.TPART	T if the part is a section of plain track
IS.DPART	T if the part is a diamond crossing
IS.PPART	T if the part is a point

Table 5.1: Projection operators and predicates for :Part.

HOL Definition 70 (Elbl.Axiom)

'Elbl = ELBSIG Join Signal | ELBL Join'

Projection operators are defined to access the join and signal field, and a predicate returning T if an edge has a signal attached is also defined.

HOL Definition 71 (ELBL_JOIN_DEF)

"(ELBL_JOIN (ELBSIG j s) = j) /\n (ELBL_JOIN (ELBL j) = j)"

HOL Definition 72 (ELBL_SIGNAL_DEF)

"ELBL_SIGNAL (ELBSIG j s) = s"

HOL Definition 73 (IS_ELBL_SIGNAL_DEF)

"IS_ELBL_SIGNAL (ELBSIG j s) = T"

Now, the basic building blocks of railway track network have been defined. The rules for building 'legal' networks will be described in the next chapter.

Chapter 6

The network model

This chapter describes the model for complete railway track networks which is specified in the HOL theory NETWORK. Networks are modelled using a class of directed graphs. Some basic properties of such network are explained.

Having created the specifications of the parts, signals and a generic graph theory, the model of a complete track network can be specified based on these building blocks. A network is modelled by a constrained, labelled directed graph whose vertices are labelled by track component parts and whose edges are labelled by joins and signals.

The basic procedures of creating a model of a track layout are:

1. construct an object of type :Part for each atomic track component with its associated track circuit—these will become the vertices;
2. construct an object of type :E1b1 for each signal and join—these will become the labels of the edges;
3. connect the adjacent parts with two antiparallel edges to represent possible traffic running in two directions—the edges are labelled by the appropriate

objects of type :Elbl.

The resulting network model is an abstract representation of the track layout. It preserves the topological relation between the adjacent parts and between the parts and signals in the original layout. The physical dimensions, such as the length of each section of track, are ignored. The specification of this model will be described in Section 6.1 and examples of track network be shown in Section 6.2. Some basic properties of the network model will be discussed in the last section of this chapter. The specification of the network model and the theorems are stored in the HOL theory `NETWORK` which is listed in Appendix A.8 and the ML source creating this theory is listed in Appendix B.9.

6.1 Specification of railway track networks

Recall that the type of a generic graph is polymorphic and is abbreviated in ML as `Graph` which stands for the type `(α)set $\#$ ($\alpha\alpha$ set)set`. An instance of `Graph` chosen to represent track layouts is defined as an abbreviated type `Network` in the HOL logic. This is possible, in contrast to the type of generic graph, because it is not a polymorphic type. The complete specification of the type `Network` is `(Part)set $\#$ (Part $\#$ Part $\#$ Elbl)set`.

The type variable `α` appeared in `Graph` is instantiated by the type `Part`, and `$\alpha\alpha$` by `Elbl`. With this type, the vertices of the network represent the track components and the edges are labelled either simply by a join or by a combination of a join and a signal. The edges represent the connection between the parts and the possible direction of the traffic moving between them.

However, not all objects of this type are proper networks according to the rules of designing track layout. A predicate **NETWORK** is required to distinguish the real railway track networks from those which, although properly typed, violate the rules. It defines a subset of the objects of type :**Network** to be proper railway track networks. This is to say that if any object of type :**Network** satisfies the predicate **NETWORK**, it is a representation of a physical track layout which can be constructed following the rules of a railway authority. The rules used in this study are taken from British Rail's current practice[54]. These rules are embedded in the definition of **NETWORK** which is defined inductively, i.e., a network can be built up by adding component parts to an existing network.

HOL Definition 74 (NETWORK.DEF)

```
"NETWORK (N:Network) =
  (P. (({ (λn. P({(HPART n)), { }) } ∧
    (λn t. P({(TPART n t)), { }) } ∧
    (λn t p n3. P({(PPART n t p n3)), { }) } ∧
    (λn t n1 n2. P({(DPART n t n1 n2)), { }) } ∧
    (λn p1 p2. (P n ∧ (p1 IS_VERTEX n) ∧
      (p1 = p2) ∧ (HFC n p1) ∧ (HFC n p2))
      ==>
      (λs1 s2. P(NJOIN n p1 s1 p2 s2))))
  ==> P n"
```

The body of this definition is an implication, which specifies that a single track component (a part) on its own is a legal network (the first four conjuncts in the antecedent), and there is only one way of building up larger networks (the last conjunct which is itself an implication). To construct a larger network, one can add a vertex into an existing network and connect this new vertex with a existing vertex by a pair of antiparallel edges. To apply this network building operation, certain conditions have to be met in order to preserve the basic network properties in the

results. These conditions are specified as the antecedent of the implication corresponding to the specification of the operation. The meanings of this specification

are:

1. the vertex p_1 with which the newly added edges are incident must be a vertex in the existing network;
2. the new vertex p_2 must not be identical to p_1 ;
3. both of them must satisfy the predicate NFC with respect to the network N .

The name NFC stands for Not-Fully-Connected. Its definition specifies that the in-degree of a vertex must be less than a limit. The maximum number of edges which are incident to a vertex depends on the kind of parts in the vertex. The limits reflect the topological characteristic of the parts. For example, at most two connections can be made to a TPART which represents a plain track because parts can only be connected to it at both ends.

HOL Definition 75 (NFC.DEF)

```
"(NFC (N:Network) (DPART n) = (IN_DEGREE N (DPART n) < 1)) /\
(NFC (N:Network) (TPART n t) = (IN_DEGREE N (TPART n t) < 2)) /\
(NFC (N:Network) (PPART n t P n3) = (IN_DEGREE N (PPART n t P n3) < 3)) /\
(NFC (N:Network) (DPART n t n1 n2) = (IN_DEGREE N (DPART n t n1 n2) < 4))"
```

Here, the out-degree of vertex is not mentioned. It has been ignored deliberately since the way a network is constructed requires that edges are added always as an antiparallel pair, which guarantees that the in-degree of each vertex is equal to its out-degree, therefore, specifying only one of them would be sufficient.

The operations carried out in the construction of a network are the general graph operations INSERT.EDGE and INSERT.VERTEX defined in the theory graph.

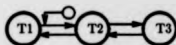


Figure 6.1: A simple network.

To make the network specification more concise, a function `NJOIN` is defined to abbreviate these graph operations.

HOL Definition 76 (NJOIN_DEF)

```

"NJJOIN (N:Network) (n1:Part) (a1:Edge) n2 a2 =
  ((a1,n2,a1) INSERT_EDGE ((a2,n1,a2) INSERT_EDGE (a2 INSERT_EDGE N)))"

```

The pre-conditions of the network construction operation `NJOIN` do not specify whether p_2 must not be a vertex of the existing network N . This implies that it may also be one of the vertices already in N . This is necessary because models for a class of layouts cannot be constructed without this lack of restriction on p_2 . This class of layouts contains one or more loops, as will be shown in the next section.

6.2 Examples of networks

Let us now study some examples of railway track networks. The first example shown in Figure 6.1 is a very simple network to illustrate the concept of Not-Fully-Connected and the placement of signals. Suppose that all vertices in this network are of `TPART`, then the middle one, namely $T2$, is fully connected, i.e., $\text{NFCT2 } N = F$, while the other two parts, $T1$ and $T3$, are not fully connected. Another point which should be mentioned here is that there is a signal attached to the edge from $T1$ to $T2$. A train moves from $T1$ to $T2$ follows this edge and is under the control of the

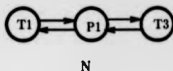


Figure 6.2: Another simple network.

signal. While a train moves along the reverse direction will follow the other edge and not be controlled by the signal.

The **second example** illustrates the operation **NJOIN**. In Figure 6.2, the network N has a not-fully-connected **PPART** $P1$ and the separate part $T2$ is a **TPART**. $T2$ is clearly not-fully-connected just after it is added into N . The conditions for constructing a larger network are satisfied so the following operation can be carried out:

NJOIN N $P1$ $j1$ $T2$ $j1$

where $j1$ has been defined to be a simple insulated join, i.e., $j1 = \text{EDGE } J_{\text{insulate}}$. The result of this is the network shown in Figure 6.3(a), and the corresponding track layout is shown in Figure 6.3(b).

The **third example** illustrates the situation in which the second vertex of the **NJOIN** operation is already in the existing network. The track layout shown in Figure 6.4(a) contains a passing loop. Suppose that a network model containing all the vertices and edges except the pair of edges labelled $j6$ has been created, and it is bound to the name N . The **NJOIN** operation can be used to insert only a pair of edges into N by taking $P12$ as the first vertex and $T4$ as the second. This can be

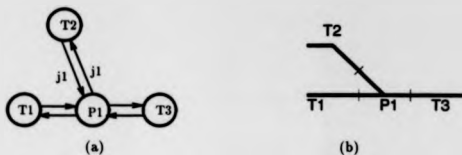


Figure 6.3: A network formed using NJOIN.

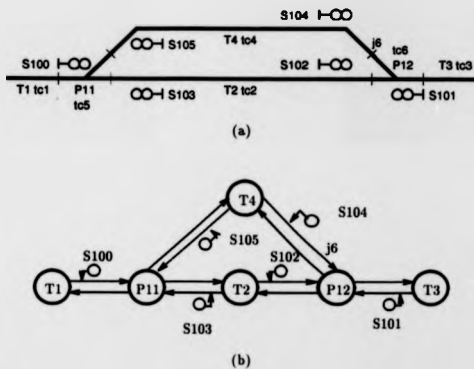


Figure 6.4: A track layout containing a passing loop.

written as

$$N' = \text{NJOIN } N \text{ P12 S104 T4 j6}$$

The effect of this operation is

$$(P12, T4, j6) \text{INSERT_EDGE}((T4, P12, S104) \text{INSERT_EDGE } N)$$

because $T4$ is already a vertex of N , the operation $T4 \text{ INSERT_VERTEX } N$ is redundant. This shows that the function NJOIN does provide the means to close a loop in a network model. Without this flexibility, it will be impossible to build any network containing a passing loop.

The complete specification of this network is:

```
{T1, T2, T3, T4, P11, P12},
{(T1,P11,S100), (P11,T1,j1), (P11,T2,j2), (T2,P11,S103),
 (T2,P12,S102), (P12,T2,j3), (P12,T3,j4), (T3,P12,S101),
 (P11,T4,j5), (T4,P11,S105), (T4,P12,S104), (P12,T4,j6)}
```

It will be shown in Chapter 7 how this network specification is verified against the generic network definition.

The last example is more extensive and realistic. It is a network model of the double left-hand junction layout shown in Figure 3.4. Its graph is shown in Figure 6.5. The specification of this network can be written in HOL in a canonical form as:

```
{ T100, T101, T102, T103, T104, T105, T106, T107, T108, T109, T110,
  T111, T112, P200, P201, D300 },
{ (T100,T101,S10), (T101,T100,j1), (T101,P200,j2), (P200,T101,j2),
  (P200,T104,j3), (T104,P200,j3), (P200,D300,j4), (D300,P200,j4),
  (T104,T105,j5), (T105,T104,j5), (T105,T106,S14), (T106,T105,j6),
  (D300,T102,j7), (T102,D300,j7), (T102,T103,S12), (T103,T102,j8),
  (T107,T108,S11), (T108,T107,j9), (T108,D300,j10), (D300,T108,j10),
  (D300,P201,j11), (P201,D300,j11), (T109,T110,S13), (T110,T109,j12),
  (T110,P201,j13), (P201,T110,j13), (P201,T111,j14), (T111,P201,j14)
  (T111,T112,S15), (T112,T111,j15) }
```


where Tn for all n appeared above has been defined as $(TPART\ n\ ncir)$, and similarly, the points $(P201, P202)$ and the diamond crossing $(D300)$ have been defined as the appropriate kind of parts. $ncir$ is the track circuit number associated with the part.

6.3 Inductive reasoning on networks

Induction is a very powerful tool of reasoning. One of the reasons that the network specification has been defined as show in **HOL Definition 74** is to take advantage of this reasoning method. The way that induction is carried out with networks is explained below.

First of all, the base cases of induction are identified. They are the single part networks. The following four theorems state that a single part of any kind is a legal network. They follow from the definition **NETWORK.DEF** immediately.

HOL Theorem 55 (NETWORK.BUFFER)

$$\vdash \forall n. \text{NETWORK}(\{\text{BPART } n\}, \{\})$$

HOL Theorem 56 (NETWORK.TRACK)

$$\vdash \forall n. \text{NETWORK}(\{\text{TPART } n\ t\}, \{\})$$

HOL Theorem 57 (NETWORK.POINT)

$$\vdash \forall n\ t\ p\ n_3. \text{NETWORK}(\{\text{PPART } n\ t\ p\ n_3\}, \{\})$$

HOL Theorem 58 (NETWORK.DIAM)

$$\vdash \forall n\ t\ n_1\ n_2. \text{NETWORK}(\{\text{DPART } n\ t\ n_1\ n_2\}, \{\})$$

The induction step is the network construction operation involving **NJOIN**. The theorem **NETWORK_NJOIN** asserts that the result of this operation is also a network providing the pre-conditions specified in the definition are met.

HOL Theorem 59 (NETWORK_NJOIN)

$$\begin{aligned} & \vdash \forall N. \text{NETWORK } N \supset \\ & \quad (\forall p_1 p_2. p_1 \text{ IS_VERTEX } N \wedge \neg(p_1 = p_2) \wedge \text{NFC } N \ p_1 \wedge \text{NFC } N \ p_2 \supset \\ & \quad (\forall e_1 e_2. \text{NETWORK}(\text{NJOIN } N \ p_1 \ e_1 \ p_2 \ e_2))) \end{aligned}$$

The induction theorem for networks is **NETWORK_INDUCT**. This states that if all the simple networks have the property P (the base cases), and if the property P holds for the results of the network building operation providing it holds for the networks being operated on (the step cases), then the property P holds for all networks. This theorem also follows from the network definition directly.

HOL Theorem 60 (NETWORK_INDUCT)

$$\begin{aligned} & \vdash \forall P. \\ & \quad (\forall n. P(\{\text{BPART } n\}, \{\}\)) \wedge \\ & \quad (\forall n \ t. P(\{\text{TPART } n \ t\}, \{\}\)) \wedge \\ & \quad (\forall n \ t \ p \ n_3. P(\{\text{PPART } n \ t \ p \ n_3\}, \{\}\)) \wedge \\ & \quad (\forall n \ t \ n_1 \ n_2. P(\{\text{DPART } n \ t \ n_1 \ n_2\}, \{\}\)) \wedge \\ & \quad (\forall N \ p_1 \ p_2. P \ N \wedge \neg(p_1 = p_2) \wedge p_1 \text{ IS_VERTEX } N \wedge \text{NFC } N \ p_1 \wedge \text{NFC } N \ p_2 \supset \\ & \quad (\forall s_1 \ s_2. P(\text{NJOIN } N \ p_1 \ s_1 \ p_2 \ s_2))) \supset \\ & \quad (\forall N. \text{NETWORK } N \supset P \ N) \end{aligned}$$

Structural induction based on this theorem can be carried out. To facilitate this in the goal directed proof style, a tactic `NETWORK.INDUCT.IAC` has been written to automate the generation of subgoals and management of the proof. The goal to which this tactic is applied should be in the following form:

```
!N. NETWORK N ==> P[N]
```

where `P[N]` is a term stating some property of `N`. It should be of type `:bool`. For example, "`!N. NETWORK N ==> GRAPH N`" is a goal in the correct form for the induction tactic. When applying the induction tactic to a properly formed goal, it generates five subgoals:

```
P[NPART...] P[TPART...] P[PPART...] P[DPART...] P[NJOIN...]
```

Suppose the network induction tactic is applied to the goal

```
"!N. NETWORK N ==> GRAPH N",
```

the following five subgoals will be generated:

```
"!N p1 p2. GRAPH N /\ ~(p1 = p2) /\ p1 IS_VERTEX N /\  
  HFC N p1 /\ HFC N p2 ==>  
  (!s1 s2. GRAPH(NJOIN N p1 s1 p2 s2))"
```

```
"!n t n1 n2. GRAPH({DPART n t n1 n2},{})"
```

```
"!n t p n3. GRAPH({PPART n t p n3},{})"
```

```
"!n t. GRAPH({TPART n t},{})"
```

```
"!n. GRAPH({NPART n},{})"
```

They corresponds to the five conjuncts in the antecedent of the induction theorem. The justification of this tactic is *modus ponens*, that is if all conjuncts of the antecedent in the induction theorem is true, then the conclusion must also be true.

6.4 Some properties of networks

One of the reasons for using Higher Order Logic in the modelling of railway signalling is its generality. This means that it is able to deduce general properties of the model such that all instances of networks created following the specification will possess the same properties. Some of the more important properties of the network model are described in this section.

Networks are graphs Although the definition of network does not explicitly specify that a network must be a graph, this is indeed always true. This is because the type `:Network` is an instance of `:Graph`, the base cases (single part networks) are trivial single vertex graphs, and all networks are built using only those graph operations that preserve the abstract property of graph. This fact is stated by the theorem `NETWORK_GRAPH`.

HOL Theorem 61 (`NETWORK_GRAPH`)

$$\forall N. \text{NETWORK } N \supset \text{GRAPH } N$$

After this fact has been established, all graph operations can be applied to networks safely, and all theorems about graphs also hold for networks. The hierarchy of the theories reflects this as well. Since the theory `graph` is an ancestor of `NETWORK`, all functions defined in the theory `graph` are available to networks and all theorems proved about graphs hold for networks as well.

Networks are finite The theorem `NETWORK_FINITE` asserts that all networks are finite, that is both the vertex set and the edge set of any network are finite.

HOL Theorem 62 (NETWORK.FINITE)

$$\vdash \forall N. \text{NETWORK } N \supset \text{FINITE}(\text{VS } N) \wedge \text{FINITE}(\text{ES } N)$$

This theorem has been proved by induction using `NETWORK.INDUCT.TAC`. The base cases are trivial. A single vertex graph is clearly finite. The results of the network building operation are finite can be deduced from the fact that adding a finite number of elements into a finite set results a finite set (more precisely, two edges are added to the edge set and possibly one vertex is added to the vertex set in each operation).

Combining the theorems `NETWORK.GRAPH` and `NETWORK.FINITE`, one can state that all networks are finite graphs.

HOL Theorem 63 (NETWORK.FINITE.GRAPH)

$$\forall N. \text{NETWORK } N \supset \text{FINITE.GRAPH } N$$

Thus, all infinite sets are excluded from networks. The practical significance of this is that there exists an upper bound on the number of components in a network. Therefore, search algorithms operated on networks should terminate eventually. This is also significant when considering the storage required for the database of the geographic data and the time required in each iteration of the control loop in the interlocking software.

Networks are connected Recall the definition `CONNECTED.DEF` in Chapter 4 which specifies that there exists a path between any two different vertices in a connected graph. This implies that no part of a connected graph is separated. The theorem `NETWORK.CONNECTED` asserts that all networks are connected.

HOL Theorem 64 (NETWORK.CONNECTED)

$$\vdash \forall N. \text{NETWORK } N \supset \text{CONNECTED } N$$

This theorem has been proved by induction using `NETWORK.INDUCT_TAC` as well. The base cases are trivial since there is only one vertex in each graph. For the induction step, what required to be proved is that by adding a vertex into a connected graph, and at the same time, adding a pair of edges connecting it to a vertex already in the existing graph, the resulting graph will still be connected. The subgoal corresponding to this is:

```
"!u p1 p2. CONNECTED u /\ ~(p1 = p2) /\
  p1 IS_VERTEX u /\ (NPC u p1) /\ (NPC u p2) ==>
  !s1 s2. CONNECTED (NJOIN u p1 s1 p2 s2)"
```

By rewriting with the definitions of `CONNECTED`, this is further divided into two subgoals. The first is in the form

```
"* * *
***
GRAPH (NJOIN u p1 s1 p2 s2)
```

where the antecedent has been abbreviated as These are essentially the subgoals corresponding to the induction step in the proof of the theorem `NETWORK.GRAPH`, which means that the result of the `NJOIN` operation must always be a graph. The same tactic used in proving the corresponding subgoal in `NETWORK.GRAPH` can be applied to solve this.

The second subgoal is also an implication, but more complex.

```
"!v1 v2.
v1 IS_VERTEX (NJOIN u p1 s1 p2 s2) /\
v2 IS_VERTEX (NJOIN u p1 s1 p2 s2) /\
~(v1 = v2) ==>
{?l.
  PATH(NJOIN u p1 s1 p2 s2) l /\
  (v1 = PATH_ENTRY 1) /\
  (v2 = PATH_EXIT 1))"
```

In essence, what required to be shown is that there exists a path between any two different vertices in the resulting network ($\text{NJOIN } p_1 \text{ } s_1 \text{ } p_2 \text{ } s_2$) given that N is connected.

To solve this, case analysis on the location of p_2 can be considered (p_1 is already a vertex in N from the definition of **NETWORK**). There are two cases:

- I. p_2 is in N ;
- II. p_2 is outside N .

In case I, the operation **NJOIN** will insert into N only two edges but no new vertex. Since N is connected, the result of adding two edges to it will clearly be connected.

The situation of case II is more complicated. Further case analysis on the locations of the variable vertices v_1 and v_2 can be considered. There are four cases according to whether v_1 and v_2 is in N :

1. both v_1 and v_2 are outside N ;
2. v_1 is in N and v_2 is outside N ;
3. v_2 is in N and v_1 is outside N .
4. both v_1 and v_2 are in N ;

The first case is trivial since there is only one vertex outside N , namely p_2 . If both v_1 and v_2 are outside N , they must both equal p_2 . This contradicts with $\neg(v_1 = v_2)$ in the antecedent of the subgoal. Case 4 is simple since N is already connected, there is a path between any two vertices in it.

The cases 2 and 3 are reciprocal with the locations of the variable vertices v_1 and v_2 transposed. The situation of case 2 is illustrated in Figure 6.6. These two

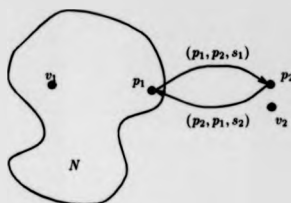


Figure 6.6: Location of vertices: case 2.

cases require further case analysis according to whether v_1 and v_2 are identical to the vertices p_1 and p_2 . There are again four cases. Taking case 2 as example, the subcase can be listed as:

- (a) $v_1 = p_1$ and $v_2 = p_2$;
- (b) $v_1 = p_1$ and $\neg v_2 = p_2$;
- (c) $\neg v_1 = p_1$ and $v_2 = p_2$;
- (d) $\neg v_1 = p_1$ and $\neg v_2 = p_2$;

There will be no path from v_1 to v_2 in cases (b) and (d) because v_2 is outside N and not equal to p_2 , which implies that v_2 is not a vertex of the resulting network. These situations contrast with the antecedent of the subgoal, so they can be solved by contradiction.

To prove each of the remaining two subcases requires an appropriate evident to be supplied to the theorem prover. The evident for subcase (a) is the single edge

path

$$[(p_1, p_2, s_1)],$$

which is one of the newly added edges connecting the two vertices. This is clearly a path in the new network. The evident for subcase (c) is the path

$$\text{APPEND } l[(p_1, p_2, s_1)]$$

where l is a path in N and $v_1 = \text{PATH_ENTRY } l \wedge p_1 = \text{PATH_EXIT } l$. Since the l is a path in N , it must be a path in a larger network containing N . Appending another path $[(p_1, p_2, s_1)]$ in the larger network should results a path providing that the conditions of combining paths specified in the theorem **PATH_APPEND** are satisfied. In this case, these conditions are satisfied because the newly inserted edge (p_1, p_2, s_1) is not equal to any of the edges in l , and the new vertex p_2 is not equal to any of the vertices passed by l . Thus, this subcase can be resolved. Case 3 can be solved using the same method.

Following the analysis, appropriate tactic can be built to solve the subgoals. thus to prove the theorem **NETWORK_CONNECTED**. The complete proof can be found in Appendix B.9.

The theorem **NETWORK_CONNECTED** is very important in practice. It implies that from any point in a network, any other point can be reached. This does not mean that a single route can be set up for a train to move between any parts in a network. Routes have not been formally defined yet.

Following the approach explained above, other general properties of the network model can be deduced. This network model and its properties provide a formal foundation on which reasoning about routes and interlocking can be carried out.

The following chapters will explain how to use the theories described in this part to help design and implement Computer Aided Design (CAD) tools and possible operational software for the signalling engineers.

Part III

Applications

In this part, three applications are presented which use the formal model of railway track network described in Part II. They are:

1. verification of track layout;
2. generation of control table;
3. interlocking of routes.

Each of these will be described in a separate chapter. The first two applications are in the area of CAD tools for signalling scheme design. The last one concerns the modelling of the logical operations of interlocking systems.

These applications can be viewed as a case study of applying the theories into practice. In the concluding chapter of the thesis, discussions of further use of the theories will be given.

Chapter 7

Verification of track layout

This chapter describes a railway track layout verifier. It accepts specifications of track layouts generated by CAD tools and verifies them against the formal model of track network. It deduces a theorem for each specification asserting the conformity of the specification to the formal network model if this is true.

As described in Section 3.3.2, the first step in the design of a signalling scheme is the specification of the track and signal layout. The result of this step is a specification for the required layout. The process of producing this specification usually involves designers and engineers from many disciplines. One of the tasks of the signalling engineers is to ensure that the new scheme will conform to all safety regulations. In order to apply rigorous methods in later stages of the design and implementation of signalling schemes, in addition to the traditional layout drawings and descriptions in natural language, a formal specification of the layout is indispensable. Formal reasoning can be carried out using this abstract representation. First of all, verification of this formal specification of track layout against the formal model of track network should be performed. This ensures that the specification conforms to the rules of creating track network, and hence it is a representation of

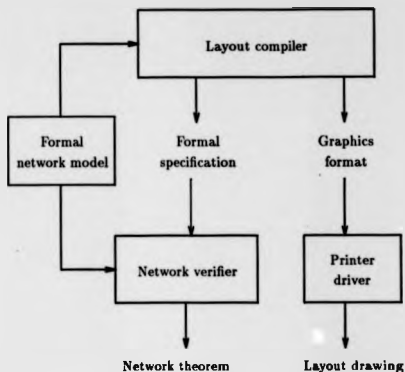


Figure 7.1: Generation and verification of track layout.

a 'legal' layout. Here, legal means conforming to the formal definition of a track network. Figure 7.1 illustrates the process of generating and verifying a formal specification. The layout compiler will be described in Section 7.1 while the subject of this chapter, the network verifier, will be described in detail in the subsequent sections.

7.1 Layout compiler

Although the subject of this chapter is the verifier, some comments on the layout compiler will help to understand the motivation, necessity and usefulness of the verifier. The layout compiler is a CAD tool acting as the front end for track layout design. A prototype compiler has been developed by Cullyer [25]. It consists of three parts: a graphical user interface, an input checker and a compiler.

The graphical user interface handles the interaction between the designer and the computer system. It shows the track layout in a symbolic form which resembles the conventional drawings, and allows the user to insert, delete or modify objects in the layout. The user interface utilizes the interactive graphics capabilities which have become standard features of personal computers and workstations. The entire layout is conceptually divided into a grid of cells. Each cell contains only a single track component. The internal representation of the layout and the rules for checking input are based on the formal track network model described in Chapter 6. When the user inserts a component into a cell, the input checker validates the component using the rules defined in the formal model. For example, a point (or *PPART*) can have at most three pairs of connections to the adjacent cells. The input checker also displays a list of components which are 'legal' in the current cell via the graphical interface to help the user choose the correct one. When the layout is completed, the compiler translates the internal representation to a formal specification in the format to be described in Section 7.3.

The layout compiler can also generate files in another format, known as *Railway Layout Graphics* language (RLG for short). This format has been defined based on

the internal representation of the layout compiler, and it is graphics output oriented. Files in RLG format are used to produce hard copy of the layout, such as the one shown in Figure 3.4.¹

Certainly, formal software development methods, such as structured system design and static code analysis, can be applied in the development of the layout compiler. However, current technologies are still not capable of verifying the correctness of the complete compiler due to the complex interactions between the graphics libraries and system libraries involved. Therefore, verification of the formal specification generated by the compiler is necessary.

7.2 The network verifier

The task of the network verifier is to take the formal specification generated by the layout compiler as its input, and to deduce a theorem asserting that the specification is an instance of a legal network according to the definition of the network model if and only if this is true.

This approach represents an isomorphism between the conventional engineering design and formal theorem proving. A commutative diagram illustrating this isomorphism is shown in Figure 7.2. In the diagram, a downward arrow indicates an abstraction, while an upward arrow indicates an interpretation. The upper path shows the conventional design process. To approve a layout scheme, the designers and engineers perform a large amount of checking against the current regulations

¹A PostScript printer driver which accepts files in RLG format has been implemented by the author. The railway track diagrams throughout this thesis have been produced by this driver.

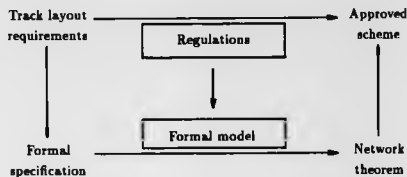


Figure 7.2: Isomorphism between engineering design and theorem proving.

and their experience, maybe with some assistance from CAD tools. The lower path shows the use of a verifier based on the HOL theorem prover. This process works on an abstract model of the real layout. It verifies the specification against the formal model of generic networks. The result is a theorem stating that the specification is an instance of generic networks.

This lower path based on formal reasoning can be used to replace a large proportion of the manual process in the upper path, and help the engineers to improve their designs, but it will never completely replace the upper path. Caution must be applied when interpreting the theorems because they are deduced based on the formal specification rooted in the generic theory, both of which have been derived by abstracting the engineering drawings and regulations. These descriptions are only as good as the model designer's understanding of the real world. However, abstraction is a very powerful tool. By creating an abstract model, the designer can gain greater insight into the problem.

There is an analogy with hardware verification. A VLSI device fabricated on

a silicon wafer will never be verified, even if low level formal proofs have been carried out. What can be verified is the formal design specification. However, this does not mean that formal verification does not have a significant role, since it helps to discover many design errors and misunderstandings. As hardware and system become even more complex, formal specification and verification will be more important.

The use of the verifier is simple. It appears to the user as an ML function in a HOL library. It takes the file name of the formal specification file generated by the compiler as its sole argument and returns a value of type `:thm` if verification is successful. Suppose that a specification of a layout has been saved in a file named `layout.rls`, the session below shows how it is verified:

```
# load_library 'rail_verifier';  
() : void  
  
# verify 'layout.rls';  
  
|- NETWORK {...}, {...}  
  
: thm
```

The library `rail_verifier` is first loaded into HOL. Then, the verifier is called with the name of the file containing the specification. If this specification conforms to the formal network model, a theorem is returned. Otherwise, the evaluation fails. The verifier automates and encapsulates the difficult, and sometimes very tedious, process of discovering a proof for each specification. This provides an easy-to-use tool to the signalling engineers.

7.3 Formal specification of track layout

Formal specifications of track layouts are written in a language called the *Railway Layout Specification Language*, RLS for short. It is the target language of the layout compiler and the source language of the network verifier. The RLS language is based on the formal model of track components and networks. The syntax and semantic of RLS will be described in separate subsections below.

7.3.1 Syntax

The syntax of the RLS language is defined in an augmented BNF form in Figure 7.3.

The following rules are used in the syntax definition:

1. all non-terminal symbols are in lower case characters;
2. all terminal symbols appear as literal character strings enclosed in single quotes except the end of specification marker which is a single character indicated as [EOF] meaning the end of file;
3. the start symbol is *spec*.

7.3.2 Semantics

A complete layout specification is divided into two parts. The first is the *definition* part which begins with the keyword **DEFINITION** and it is ended by the start of the second part which is introduced by the keyword **CONSTRUCTION**. The construction part extends to the end of the file.

```

spec ::= definition_part construction_part [EOF] ;;

definition_part ::= 'DEFINITION' def_list ;;

def_list ::= definition def_list ;;

definition ::= 'TCIR' num | 'BPART' num |
'TPART' num num |
'PPART' num num num '(' num num num ')' |
'BPART' num num '(' num num ')' '(' num num ')' |
'SIGNAL' num sig_type | 'POINT' num |
'EDGESIG' num join_type num | 'EDGEJOIN' num join_type ;;

join_type ::= 'CONDUCT' | 'INSULATE' | 'OVERLAP' | 'TERMINATE' ;;

sig_type ::= 'MAIN' | 'MAIN_JUNC' | 'MAIN_SUB' |
'MAIN_SUB_JUNC' | 'SHUNT' ;;

construction_part ::= 'CONSTRUCTION' 'SIMP' part op_list ;;

op_list ::= op op_list;;

op ::= 'EJOIN' part part edge edge |
'EDGE' part part edge edge ;;

part ::= 'B'num | 'T'num | 'P'num | 'D'num ;;

edge ::= 'j'num | 's'num ;;

num ::= digit | digit num;;

digit ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;;

```

Figure 7.3: Syntax of Railway Layout Specification language.

The definition part The DEFINITION part contains definitions of all the track components, signals and joins appeared in the layout. Each definition associates an identification number to the object being defined and specifies the types or other sub-objects required to fully define such an object in the HOL logic.

There are nine different definitions allowed in the definition part. The keywords which introduce the definitions and the meanings of their associated fields are listed below:

TCIR(C) Track circuit. The only field is the track circuit number.

POINT(N) Point. The only field is the ID number.

BPART(B) Buffer part. The only field is the part ID number.

TPART(T) Plain track part. The first field is the part ID number and the second is the ID number of the associated track circuit.

PPART(P) Point part. The first two fields have the same meaning as **TPART**. The third field is the ID number of the point which must be defined previously by a **POINT** definition. The three field enclosed in parentheses are the ID numbers of the trailing, normal and reverse parts, respectively.

DPART(D) Diamond crossing part. The first two fields have the same meaning as **TPART**. The two pairs of num fields are the ID numbers of the adjacent parts. Each pair identifies the parts connecting to the same leg.

SIGNAL(S) Compound signal. The first field is the ID number and the second is the class.

EDGEJOIN(J) Simple edge label. The first field is the ID number, the second the type of join.

EDGESIG(s) Edge label containing a signal. The first two fields have the same meaning as **EDGEJOIN**. The last field is the ID number of the attached signal which has been defined by a **SIGNAL** definition.

The action for each definition is to define a constant in the HOL logic. The name of the constant is the ID number prefixed by a single letter indicating the type of the object. The prefix letters for each type of object is enclosed in parentheses following the keywords in the list above. All the information necessary for creating constants of various types is provided by the fields in the definition. Note that all of the state functions of circuits, points and signals have been omitted. This is because, when verifying the layout, the states of these components are not important; only the static topological relations between the components are being considered. Dummy functions of the appropriate types are supplied by the verifier to satisfy the type checker when defining logical constants.

The construction part This part contains information about how to build the network using the objects defined in the definition part. This information is necessary to guide the verifier in the deduction of the network theorem. It appears as a list of network operations. Each operation adds some objects into the network built by previous operations in the list except the first one. The first operation must be a **SIMP** operation which means to construct a simple network containing only a single part. This corresponds to the base case in the definition of network **NETWORK_DEF**. In addition to the **SIMP** operation, there are two more operations allowed:

NJOIN adds a vertex and a pair of edges into the network. The first part indicates the vertex in the existing network to which new connections are being made.

The second part is a new vertex to be inserted into the network. The two edge fields specify the labels of the edges connecting the two parts.

EDGE adds only a pair of edges. The fields in this operation have the same meaning as in NJOIN except that the second part is also a vertex in the existing network.

For each operation, the verifier attempts to prove a theorem in the form:

NETWORK (NJOIN N p_1 p_2 e_1 e_2)

where N is the network built by the operations so far, p_1 and p_2 are the parts, e_1 and e_2 are the edge labels. If the attempt fails, the operation violates the rules of the formal network model, thus no theorem can be deduced.

Example A specification of the layout of a passing loop shown in Figure 6.4 can be written as below:

```

DEFINITION
TCIR 1 TCIR 2 TCIR 3 TCIR 4 TCIR 5 TCIR 6
POINT 11 POINT 12
TPART 1 1
PPART 11 5 1 (1 2 4)
TPART 2 2
PPART 12 6 2 (3 2 4)
TPART 3 3
TPART 4 4
EDGEJOIN 1 INSULATE
SIGNAL 100 MAIN SIGNAL 101 MAIN SIGNAL 102 MAIN
SIGNAL 103 MAIN SIGNAL 104 MAIN SIGNAL 105 MAIN
EDGEISIG 100 INSULATE 100
EDGEISIG 101 INSULATE 101
EDGEISIG 102 INSULATE 102
EDGEISIG 103 INSULATE 103
EDGEISIG 104 INSULATE 104
EDGEISIG 105 INSULATE 105
CONSTRUCTION
SIMP T1
NJOIN T1 P1 #100 j1
NJOIN P11 T2 j1 #103

```

```
HJOIN T2 P12 s102 j1
HJOIN P12 T3 j1 s101
HJOIN P11 T4 j1 s105
EDGE T4 P12 s104 j1
[EOF]
```

Note that different lists can create identical networks, since the order of the operations in the construction part is not unique.

7.4 The implementation of the verifier

The verifier can be divided into two parts: the parser and the prover. The parser is the front end of the verifier. Its functions are to recognize the input and to call appropriate functions in the prover. The prover is a suite of ML functions which together carry out proofs and deliver theorems if the specification is correct according to the formal model defined in the network theory.

7.4.1 The parser

The parser is implemented using the parser generator in the standard collection of HOL libraries. This parser generator accepts grammar specification with embedded actions in a syntax similar to the BNF. It generates a parser in the form of an ML function, named `PARSE_file`. When this function is called, it will read the input, and attempt to match the production rules specified as the syntax of the RLS language in Figure 7.3. If a production rule is matched, the associate action is invoked. This parser function is called by the function `verify` mentioned in the example at the end of Section 7.2 as the entry point to the parser.

To illustrate the implementation of the parser, two production rules with associ-

ated actions as part of the input grammar specification to the parser generator are described in detail below. The complete ML source of the verifier can be found in Appendix C.

The first rule is the plain track definition rule. The syntax for a plain track part is `TPART num num` (see Figure 7.3). The production rule for parsing this syntax required by the parser generator is written as:

```
tpart --> [TPART] {def_tpart(TOKEN, TOKEN)}.
```

This specifies that the definition starts with a literal string 'TPART'. When the parser sees such a string, it will read the next two tokens from the input stream and pass them to the function `def_tpart`. This function is defined as below:

```
let def_tpart (id, tc) = % (string 0 string) -> (string list 0 thm) %
  if ((is_num id) & (is_num tc))
  then
    (let pname = 'T' ^ id in
     let tcir = mk_const(('C' ^ tc), ":Tcir") in
     let t = mk_eq(mk_var(pname, ":Part"),
                  mk_comb(mk_comb("TPART", (mk_num id)), tcir)) in
     ([pname], new_definition(pname, t)))
  else failwith "expecting two numbers as ID's (def_tpart)";;
```

It validates the tokens by calling the function `is_num` which returns true if the token string contains only digits. If the tokens are valid, it proceeds to create a new logical constant for this plain track part. Suppose that the tokens are the strings '123' and '201', the effect of evaluating `def_tpart` is equivalent to making the following HOL definition:

HOL Definition 77

```
"T123 = TPART 123 C201"
```

where `C201` has been defined as a track circuit with ID number 201. Actions associated with other production rules in the definition part carry out similar definitions.

The next production rule to be described is the `NJOIN` construction rule. The syntax of this rule is `NJOIN part part edge edge`. The input to the parser generator is written as below:

```
njoin --> [NJOIN] {mk_njoin(part_nums, edge_nums)}
```

The function `mk_njoin` is defined as below:

```
let mk_njoin (([pt1; pt2], t1:thm), ([ed1; ed2], t2:thm))
  % : (string list # thm) # (string list # thm) -> %
  : (string list # thm) =
  let p1 = mk_const(pt1, ":Part") in
  let p2 = mk_const(pt2, ":Part") in
  let e1 = mk_const(ed1, ":Elbl") in
  let e2 = mk_const(ed2, ":Elbl") in
  let th = prove_network_njoin rail_tmp_thm p1 p2 e1 e2 in
  ([pt2], (rail_tmp_thm := save_thm((pt2~'TH'), th))) ;;
```

The arguments `pt1` and `pt2` are the parts to be connected and the arguments `ed1` and `ed2` are the edge labels. They have been validated by the production rules `part_nums` and `edge_nums`. The global identifier `rail_tmp_thm` is bound to the theorem returned as the result of the previous step of building network. The function `prove_network_njoin` in the prover is called to deduce a theorem for the current step. If this succeeds, the new theorem is saved in the current theory and also bound to `rail_tmp_thm` to pass to the next step. This process continues until the entire specification file is exhausted.

7.4.2 The prover

The prover automates the process of proving theorems about instances of networks. The theorems it deduces are in the form

$$\vdash \text{NETWORK}(\{\dots\}, \{\dots\}) \quad (7.1)$$

where the actual element of the vertex set and edge set have been abbreviated. This theorem asserts that $(\{\dots\}, \{\dots\})$ is an instance of a generic network as defined in **NETWORK_DEF**. There are three ML functions at the top level which are called by the parser action functions. The function **prove_simple_network** delivers a theorem as an instance of the general theorem **NETWORK_SIMP**. This corresponds to the base cases of the network induction theorem. The other two functions, **prove_network_njoin** and **prove_network_edge**, deduce theorems which are instances of the network induction step theorem **NETWORK_NJOIN**. The difference between these two functions is that **prove_network_njoin** takes advantage of the fact that the second operand of the **NJOIN** operation is not a vertex in the network so the prove is simpler. The proof strategy used by these two function is *modus ponens* with the theorem **NETWORK_NJOIN**. The function **prove_network_njoin** is listed in Figure 7.4 and it is described in detail below.

To deduce a theorem of the following form

$$\text{NETWORK}(\text{NJOIN } N \ n_1 \ n_2 \ s_1 \ s_2), \quad (7.2)$$

one can prove the antecedent of the theorem **NETWORK_NJOIN**, then apply *modus ponens* rule. Taking N as the network created so far, the theorem $\vdash \text{NETWORK } N$ will be returned as the result of the previous production rule and supplied to **prove_network_njoin** as its first argument **thm1**. Using *modus ponens* rule with **NETWORK_NJOIN** and **thm1** results in an implicative theorem 7.3:

$$\begin{aligned} &\vdash \forall p_1 \ p_2. \ p_1 \text{ IS_VERTEX } N \wedge \neg(p_1 = p_2) \wedge \text{NFC } N \ p_1 \wedge \text{NFC } N \ p_2 \supset \\ &(\forall e_1 \ e_2. \ \text{NETWORK}(\text{NJOIN } N \ p_1 \ e_1 \ p_2 \ e_2)) \end{aligned} \quad (7.3)$$

Since the conclusion of this theorem matches 7.2, *modus ponens* rule can be used

```

1 let prove_network_njoin thm1 p1 p2 j1 j2 =
2   let p.n1 = (dest_comb (concl thm1)) in
3   if (not('NETWORK' = (fst (dest_const p)))) then
4     failwith 'not NETWORK theorem' else
5     let lm = (SPEC ""n1" NETWORK_NJOIN) in
6     let thm2 = prove_in_network p1 n1 in
7     let thm3 = EQF_ELIM (Part_EQ_CONV ""p1 = "p2") in
8     let thm4 = prove_SFC p1 n1 in
9     let thm5 = prove_not_in_network p2 n1 in
10    let thm5 = NP (NP (SPEC [n1;p2] NOT_VER_INF_SFC) thm1) thm5' in
11    let ante = CONJ thm2 (CONJ thm3 (CONJ thm4 thm5)) in
12    let lm' = SPEC [p1;p2] (NP lm thm1) in
13    let network_canon thm =
14      let njointhm = NP (SPEC [n1;p1;j1;p2;j2] NJOIN_EIP)
15        (CONJ thm2 thm5') in
16      let th = PURE_ONCE_REWRITE_RULE[VERTICES;EDGES]
17        (PURE_ONCE_REWRITE_RULE[njointhm] thm) in
18      (CONV_RULE (DEPTH_CONV (UNION_CONV Part_EQ_CONV)) th) in
19    let nth = (SPEC [j1; j2] (MATCH_NP lm' ante)) in
20    network_canon nth ?
21    failwith 'prove_network_njoin';
22

```

Figure 7.4: Listing of prove_network_njoin.

again to deduce a theorem in the form of 7.2 if the four conjuncts of the antecedent of 7.3 can be proved. The strategy for solving these four subgoals is as follows (the line numbers refer to the listing in Figure 7.4):

Subgoal 1 $n_1 \text{ IS_VERTEX } N$. The function `prove_in_network` is used to prove this subgoal (line 6). It returns a theorem matching the subgoal if n_1 is a vertex of N . It uses the conversion `IN_CONV` in the `sets` library which returns a theorem $\vdash x \text{ IN } \{x_1, \dots, x_n\} = \top$ if and only if x is equal to x_i for some i where $1 \leq i \leq n$.

Subgoal 2 $\neg(n_1 = n_2)$. This subgoal is proved using the conversion `Part_EQ_CONV` (line 7). This conversion returns a theorem $\vdash (p_1 = p_2) = \top$ if and only if p_1 and p_2 are syntactically equal or all their sub-fields are equal. Otherwise, the theorem $\vdash (p_1 = p_2) = \text{F}$ is returned. The function `EQF_ELIM` transforms a theorem $\vdash t[x] = \text{F}$ to $\neg t[x]$.

Subgoal 3 $\text{NFC } N \ n_1$. This subgoal is proved using the function `prove_NFC` (line 8) which returns a theorem of the form $\vdash \text{NFC } N \ p_1$ if the vertex p_1 is not fully connected.

Subgoal 4 $\text{NFC } N \ n_2$. The last subgoal is proved using the fact that p_2 is not a vertex of N . The function `prove_not_in_network` (line 9) returns a theorem of the form $\vdash \neg(p_2 \text{ IS_VERTEX } N)$. Then, *modus ponens* is applied to the theorem `NOT_VER_IMP_NFC` to deduce $\vdash \text{NFC } N \ p_2$ (line 10).

Details of the lower level functions mentioned above can be found in Appendix C.4. The *modus ponens* rule is applied to the conjunction of the above four subgoals and an instance of 7.3 to deduce a theorem in the form of 7.2 (line 19). This theorem is

again to deduce a theorem in the form of 7.2 if the four conjuncts of the antecedent of 7.3 can be proved. The strategy for solving these four subgoals is as follows (the line numbers refer to the listing in Figure 7.4):

Subgoal 1 $n_1 \text{ IS_VERTEX } N$. The function `prove.in.network` is used to prove this subgoal (line 6). It returns a theorem matching the subgoal if n_1 is a vertex of N . It uses the conversion `IN_CONV` in the `sets` library which returns a theorem $\vdash x \text{ IN } \{x_1, \dots, x_n\} = T$ if and only if x is equal to x_i for some i where $1 \leq i \leq n$.

Subgoal 2 $\neg(n_1 = n_2)$. This subgoal is proved using the conversion `PART_EQ_CONV` (line 7). This conversion returns a theorem $\vdash (p_1 = p_2) = T$ if and only if p_1 and p_2 are syntactically equal or all their sub-fields are equal. Otherwise, the theorem $\vdash (p_1 = p_2) = F$ is returned. The function `EQP_ELIM` transforms a theorem $\vdash t[x] = F$ to $\neg t[x]$.

Subgoal 3 $\text{NFC } N \ n_1$. This subgoal is proved using the function `prove.NFC` (line 8) which returns a theorem of the form $\vdash \text{NFC } N \ p_1$ if the vertex p_1 is not fully connected.

Subgoal 4 $\text{NFC } N \ n_2$. The last subgoal is proved using the fact that p_2 is not a vertex of N . The function `prove.not.in.network` (line 9) returns a theorem of the form $\vdash \neg(p_2 \text{ IS_VERTEX } N)$. Then, *modus ponens* is applied to the theorem `NOT_VER_IMP_NFC` to deduce $\vdash \text{NFC } N \ p_2$ (line 10).

Details of the lower level functions mentioned above can be found in Appendix C.4. The *modus ponens* rule is applied to the conjunction of the above four subgoals and an instance of 7.3 to deduce a theorem in the form of 7.2 (line 19). This theorem is

then converted into a canonical form by expanding the operation NJOIN using the function `network.canon` (line 20). The resulting theorem is in the form matching 7.1 as shown below

$$\text{NETWORK}(\{\dots, n_2\}, \{\dots, (n_1, n_2, s_1), (n_2, n_1, s_2)\})$$

where the new vertex and the new edges are added to the vertex set and edge set of the network, and they are in pure set syntax and free of any network operators.

To conclude this chapter, the verification of the specification of the passing loop layout listed on page 116 in Section 7.3, with some intermediate results, is shown in Figure 7.5. This session shows how the network expands when new vertices and edges are added. It also shows that there are large number of intermediate theorems being generated in each step: the total number of intermediate theorems is 28419. The verifier does integrate the theorem proving process and makes theorem prover easier to use. The timing information is obtained in a Sun3 with 12Mbyte of memory. The time can be further reduced since the current implementation has not been optimized. The final result is the network theorem in its canonical form.

```

$ verify 'loop';

(['T1'], |- NETWORK({T1},{})
: (string list $ thm)
Run time: 0.0s
Intermediate theorems generated: 1

(['P11'], |- NETWORK({P11,T1},{(T1,P11,s100),(P11,T1,j1)}))
: (string list $ thm)
Run time: 11.7s
Garbage collection time: 21.2s
Intermediate theorems generated: 1179

(['T2'],
|- NETWORK
  { (T2,P11,T1),{(P11,T2,j1),(T2,P11,s103),(T1,P11,s100),(P11,T1,j1)} })
: (string list $ thm)
Run time: 25.7s
Garbage collection time: 54.3s
Intermediate theorems generated: 2288

... (deleted)

(['P12'],
|- NETWORK
  { (T4,T3,P12,T2,P11,T1),
    { (T4,P12,s104),(P12,T4,j1),(P11,T4,j1),(T4,P11,s106),(P12,T3,j1),
      (T3,P12,s101),(T2,P12,s102),(P12,T2,j1),(P11,T2,j1),(T2,P11,s103),
      (T1,P11,s100),(P11,T1,j1)} })
: (string list $ thm)
Run time: 126.1s
Garbage collection time: 230.3s
Intermediate theorems generated: 10020

```

Figure 7.5: A HOL session of verifying network specification.

Chapter 8

Generation of control tables

This chapter describes a method for automatic generation of control tables. This method utilizes well-known graph search algorithms to find out routes in a network and then works out the entries in the control table based on the formalized interlocking regulations.

The second step in the design of an SSI controlled signalling scheme is the generation of control tables, whose format has been shown in Chapter 3. Information is extracted from the layout and filled into the control tables. This is probably the most important step in the design process, with respect to ensuring the safe operation of the system. If any incorrect information is left in the table unnoticed, erroneous geographic data would be generated, and the interlocking may become unsafe. It is therefore, necessary to automate the generation of control tables by employing verified software and then to formally verify of the contents of the resulting tables.

Since all the topological information about a layout is encompassed in its formal specification, the data required to fill the control table can be extracted from this specification. This chapter describes a method for finding routes, deriving information to fill control tables and verifying the results. This method uses the formal specification of a layout generated by the layout compiler and verified by the verifier.

The definition of routes is described in Section 8.1 and the algorithms for finding routes are discussed in Section 8.2. The last section of the chapter describe how to generate and verify the control tables.

8.1 Definition of routes

In Section 3.2, a route is described as a section of railway track starting and ending at signals. In the framework of the formal network model, a route is a path starting at a signal and terminating at another signal, but extra rules are required to restrict the paths which can be considered as routes. These rules are due to the physical nature of the track component parts. They are:

1. when passing a point, i.e., a PPART, a route cannot enter from a normal edge and continue to a reverse edge or vice versa;
2. when passing a diamond crossing, i.e., a DPART, a route cannot move from an edge of one leg to an edge of another leg.

Referring to the example layout of double left-hand junction shown in Figure 3.4, a train cannot move from T104 to D300 through P200 following a single route, nor can a train move from T108 to P200 through D300 in a single manoeuvre.

A route from the signal S10 to the signal S12 is the following list of edges:

```
S10S12 = [  
  (T100, T101, S10); (T101, P200, J2);  
  (P200, D300, J4); (D300, T102, J7); (T102, T103, S12) ]
```

By convention, the name of a route is the string formed by concatenating the names of the entry signal and the exit signals. Notice that the entry of the route is the

vertex before the entry signal, and the exit of the route is the vertex after the exit signal.

Based on the above discussion, a predicate ROUTE is defined in the HOL logic to specify what a route is.

HOL Definition 78 (ROUTE_DEF)

```
"ROUTE (N:Network) (r:(Part@Part@Sbl)list) =
  (NETWORK N) /\ (PATH N r) /\ (ROUTE_TAIL r) /\
  (IS_ELBL_SIGNAL (elb (HD r))) /\ (IS_ELBL_SIGNAL (elb (LAST r)))"
```

A list of edges r is a route in the network N if and only if it is a path in N , and it satisfies the predicate ROUTE.TAIL which is defined below, and both its first and last edges have a signal attached.

HOL Definition 79 (ROUTE_TAIL_DEF)

```
"(ROUTE_TAIL [] = T) /\
 (ROUTE_TAIL (CONS (h:Part@Part@Edge) t) =
  (t = []) /\
  (((IS_PPART (e_dec h)) =>
    ((TRAILING_EDGE h) =>
      (NORMAL_EDGE (HD t) /\ REVERSE_EDGE (HD t)) |
      TRAILING_EDGE (HD t)) |
    ((IS_DPART (e_dec h)) => (SAME_LEG h (HD t) | T)) |
    (ROUTE_TAIL t))))"
```

The function ROUTE.TAIL verifies whether the two restrictions listed at the beginning of this section are satisfied by the list of edges. The meanings of the functions used in its definition have been given in Table 5.1.

Note that, in some cases, a route may not terminate at a signal, for example, the last section at a terminal station will end with a buffer, and sometimes a partial route which does not start from a signal nor ends at a signal may be required for certain manoeuvre. These are considered to be a *subroute*. Predicate can be defined

for subroutes which will be syntactically identical to the definition of `ROUTE` except the last two conjuncts are absent. In the following discussion, only complete routes are being considered.

The functions `ROUTE.EDGES` and `ROUTE.PARTS` are defined for extracting edges and parts along the route. They are used in the specifications described in Section 8.3.

HOL Definition 80 (`ROUTE.EDGES_DEF`)

```
"ROUTE_EDGES (r:(Part#Part#Elbl)list) = (BUTLAST r)"
```

Given a route r as its argument, the function `ROUTE.EDGES` returns a list of all the edges through which r passes except the last one. A train cannot pass through this edge if the signal attached to it is ON, therefore, when considering the required points for a route, this edge can be ignored.

HOL Definition 81 (`ROUTE.PARTS_DEF`)

```
"ROUTE_PARTS (r:(Part#Part#Elbl)list) = VER_LIST (BUTLAST (TL r))"
```

The function `ROUTE.PARTS` returns a list of the parts through which the route r passes, except the part in the source vertex of the first edge and the part in the destination vertex of the last edge. The former is before the entry signal and the latter is ahead of the exit signal, therefore, they should not be considered to be in the route.

If two routes share one or more parts, they are said to be *conflicting routes*. This property is modelled by the predicate `CONFLICTING.ROUTES`.

HOL Definition 82 (`CONFLICTING.ROUTES_DEF`)

```
"CONFLICTING_ROUTES (N:Network) x1 x2 =  
  (ROUTE N x1) /\ (ROUTE N x2) /\  
  ~(DISJ_LIST (ROUTE_PARTS x1) (ROUTE_PARTS x2))"
```


This predicate is true if and only if both r_1 and r_2 are routes in the network N and their vertex lists are not disjoint. For example, in the network shown in Figure 3.4, the routes S10S12 and S11S16 are conflicting routes because they share the vertex D300, i.e.,

$$\text{CONFLICTING_ROUTES } N \text{ S10S12 S11S16} = T.$$

8.2 Finding routes

After defining what routes are, one can proceed to search for routes in a network. Since networks are graphs, many well-known graph searching algorithms can be used to find routes in a network. Amongst them, two are very suitable for finding routes: Murchland's all paths algorithm and depth-first search algorithm. However, these algorithms need to be augmented, because routes are not merely paths, there are extra constraints as specified by ROUTE.TAIL.DEF.

Murchland's algorithm[53][56] finds all possible paths between two specific vertices. An algebra is used to describe the paths in this algorithm. A path consists of the edge a followed by the edge b is written as ab . If there are two paths ab and cd between the vertices v and w , then they can be written as $ab + cd$. To avoid looping, an expression, such as $(abc)(ade)$, which has a common factor is defined to be 0. These expressions can well be represented by a tree. For computational purpose, the actual algorithm is described with the aid of an $n \times n$ matrix where n is the number of vertices. The elements of the matrix represent the path or paths between any two given vertices. Initially, the elements contain only single edge path connecting adjacent vertices. On each iteration, the expressions representing the current

known paths between the starting vertex and any other vertices are updated by adding new edges to extend the paths. When the algorithm terminates, the elements corresponding to the path(s) between the starting and ending vertices will contain an expression representing all the possible paths. To adopt this algorithm for finding routes, tests corresponding to the specification in `ROUTE.TAIL.DEF` should be incorporated in the iteration to pick out the appropriate edges. This algorithm is suitable for networks which contain larger number of possible routes between the specific vertices.

The depth-first search algorithm is described in many textbooks of elementary graph theory and of computer algorithms. Aho *et al* in [1] give a concise description of the algorithm in imperative programming style, while Paulson in [55] demonstrates an effective implementation of the algorithm in a functional programming language — Standard ML. When adopting this algorithm, tests corresponding to the specification in `ROUTE.TAIL.DEF` should also be incorporated in the search procedure to avoid going down the illegal edges.

A program implementing a suitable algorithm can be developed without too much difficulty. It will read in a specification of a network and generate a list of all possible routes in this network. Using a method similar to the one described in Chapter 7, the list of routes can be verified to show the conformity to the specification of `ROUTE.DEF`. The discussion in the next section assumes such a route finding program exists, and that it produces a list of all possible routes.

8.3 Automatic control table generation

The problem of generating control tables can now be considered. The first task is to codify the safety rules which specify the interlocking requirements for setting up a route. Considering only the simple and most common situations, the safety requirements for setting up a route [54] are the followings:

1. all track circuits on the route must be clear (column 2);
2. all points on the route must be set, locked and detected at the correct position according to the travelling direction of the route (column 4 and 5);
3. the exit signal must be in working order, i.e., showing either ON or OFF aspect (column 6);
4. the entry signal to the conflicting routes must be proved ON (column 7);
5. the track circuits from the entry signals of the conflicting routes to the point of conflict must be clear (column 2).

The column numbers enclosed in parentheses refer to the control table shown in Table 3.2. In that table, column 1 lists the name of routes. Column 2 lists the track circuits required by the route, and is corresponding to the requirement specified by rule 1 and 5. The correspondence between the other columns and the rules is as indicated above.

Each of these requirements can be specified in HOL by a function which returns a list of objects that are required by the rule. A null list indicates that nothing is required by the route.

Rule 1 The specification for rule 1 is TCIRCUITS.

HOL Definition 83 (TCIRCUITS_DEF)

```
"TCIRCUITS (r:(Part#Part#K1b1)list) =
  MAP PART_CIRCUIT (ROUTE_PARTS r)"
```

Since ROUTE_PARTS returns a list of all parts in the route, the higher-order function MAP applies PART_CIRCUIT to these parts, the result of this function is a list of all track circuits in the route.

Rule 2 This rule can be specified by two functions: NORM.POINTS which returns a list of points required NORMAL and REV.POINTS which returns a list of points required REVERSE.

HOL Definition 84 (NORM_POINTS_DEF)

```
"NORM_POINTS r = FLAT (MAP NORM (ROUTE_EDGES r))"
```

HOL Definition 85 (REV_POINTS_DEF)

```
"REV_POINTS r = FLAT (MAP REV (ROUTE_EDGES r))"
```

The functions NORM and REV take an edge as their argument, and return a list of points required NORMAL and REVERSE, respectively, if a movement from the source vertex to the destination vertex is made.

HOL Definition 86 (NORM_DEF)

```
"NORM (p1,p2,(e:Edge)) =
  ((IS_PPART p1) /\ (PART_PNT_NORMAL p1 = PART_ID p2)) =>
  [PART_POINT p1] | []"
```

HOL Definition 87 (REV_DEF)

```
"REV (p1,p2,(e:Edge)) =
  ((IS_PPART p1) /\ (PART_PNT_REVERSE p1 = PART_ID p2)) =>
  [PART_POINT p1] | []"
```

Rule 3 The specification for rule 3 is `EXIT.SIG` which returns the signal attached to the last edge of the list. If there is no signal attached to this edge, an empty list is the result. This situation may arise when partial route is being considered.

HOL Definition 88 (`EXIT.SIG_DEF`)

```
"EXIT_SIGNAL (x: (Part@Part@Klbl))list) =
  let e = elb (LAST x) in
  (IS_KLBL_SIGNAL e) => [KLBL_SIGNAL e] | []"
```

Finding conflicting routes The handling of rules 4 and 5 is more complicated because they require the search for conflicting routes. Since all possible routes in a network can be found by the program mentioned in Section 8.2, all conflicting routes of a given route r can be found using the function `CONFLICT_ROUTES`. It takes two arguments: the list of all routes in the network, and the given route r . `CONFLICT_ROUTES rlist r` delivers a list of routes which are in $rlist$ and are in conflict with r .

HOL Definition 89 (`CONFLICT_ROUTES_DEF`)

```
"(CONFLICT_ROUTES [] x = []) /\
 (CONFLICT_ROUTES (CONS h t) x =
  (~ (DISJ_LIST (ROUTE_PARTS x) (ROUTE_PARTS h)) /\
   ~(h = x)) =>
   (CONS h (CONFLICT_ROUTES t x)) | (CONFLICT_ROUTES t x))"
```

Rule 4 The function specifying rule 4 can now be defined. This function takes two arguments: the current route and the list of all routes in the network. It picks the entry signal from each route returned by `CONFLICT_ROUTES` using the function `ENTRY_SIG` which returns the signal attached to the first edge of the list.

HOL Definition 90 (`ENTRY_SIGNALS_DEF`)

```
"ENTRY_SIGNALS x rlist = MAP ENTRY_SIG (CONFLICT_ROUTES rlist x)"
```

HOL Definition 91 (ENTRY_SIG_DEF)

```
"ENTRY_SIG (x:(Part@Part@Hbl1)list) = let a = elb (ED x) in
  (IS_ELBL_SIGNAL a) => [ELBL_SIGNAL a] | []"
```

Rule 5 Let r be the route under consideration. The strategy of computing the list of all track circuits specified by this rule is:

1. work out a list, say $crlist$, containing all routes which are in conflict with r ;
2. for each vertex p on the route r which is either a PPART or a DPART, work out a list pil containing all routes which are in conflict with r at p , i.e., share the part in p ;
3. for each route i in pil , take the initial segment of i up to the vertex p ; the track circuits associated with the elements in this segment are those required by Rule 5.

Several auxiliary functions are needed in specifying the function for this strategy. Their names have CR_ as prefix. The first is CR_TAKE. Its specification is

$$CR_TAKE[p_1:\dots;p_i:\dots;p_n]p_i = [p_1:\dots;p_{i-1}]$$

i.e., it returns the initial segment of the list up to but not including the part given as its second argument.

HOL Definition 92 (CR_TAKE_DEF)

```
"(CR_TAKE [] (p:Part) = []) /\
  (CR_TAKE (CONS h t) p =
    (h = p) => [] | (CONS h (CR_TAKE t p)))"
```

If no element in the list is equal to p_i , CR.TAKE returns the whole list. The next function is CR.PRS, which stands for Partial RouteS, with the following specification:

$$\text{CR.PRS } p \left[[p_{11}; \dots; p_{1i}; \dots; p_{1n}] ; \dots [p_{m1}; \dots; p_{mi}; \dots; p_{mn}] \right] = \\ \left[[p_{11}; \dots; p_{1i}; \dots; p_{1n}] ; \dots [p_{m1}; \dots; p_{mi}; \dots; p_{mn}] \right]$$

where $p_{ji} = p$ for all $1 \leq j < k$. This function takes a part p and a list of lists of parts pll , it returns a list of lists of parts which are the initial segments of the argument lists pll . The local value $cr1st$ contains the lists which are in pll and have p as one of their elements.

HOL Definition 93 (CR.PRS_DEF)

```
"CR_PRS (p:Part) pll =
  let cr1st = FILTER pll (\l. (MEMB 1 p)) in
  MAP (\l. CR_TAKE 1 p) cr1st"
```

The function CR.PTS, which stands for ParTS, takes two arguments: the first is the list of parts forming the current route, and the second is a list of lists of parts which are obtained from the list of all routes in the network. It returns a list of lists of parts in the initial segments of all the conflicting routes.

HOL Definition 94 (CR.PTS_DEF)

```
"(CR_PTS [] (pll:((Part)list)list) = []) /\
(CR_PTS (CONS (p:Part) t) pll =
  ((IS_PPART p) /\ (IS_OPART p)) =>
  (APPEND (CR_PRS p pll) (CR_PTS t pll))
 | (CR_PTS t pll))"
```

Now, the top level function for Rule 5 can be specified as CR.TCIRCUITS. It takes two arguments similar to ENTRY.SIGS, the first, r is the current route and the

second *rlist* is the list of all routes in the network. It delivers a list containing all track circuits in the conflicting routes between the entry signals and the points of conflict.

HOL Definition 95 (CR.TCIRCUITS.DEF)

```
"CR.TCIRCUITS r rlist =  
  let crlst = CONFLICT_ROUTES rlist r in  
  let ptlist = FLAT (CR_PTS (ROUTE_PARTS r) (MAP ROUTE_PARTS rlist)) in  
  (MAP PART_CIRCUIT ptlist)"
```

The local value *crlist* contains all routes which are in conflict with *r*, and *ptlist* is the list of all parts formed by flattening the initial segments of all the conflicting routes returned by *CR_PTS*. A list of track circuits is obtained by applying *PART_CIRCUIT* to *ptlist*.

The specification of the rules has now been written as HOL functions. These functions form the core of the specification of a program which generates the control tables. Such a program can be implemented in a verifiable subset of high-level programming language, such as SPADE-PASCAL[11] or SPARK Ada[12]. If the program is verified and validated to correctly implement the safety rules, the control tables generated by it should be free of errors. However, it is still possible to verify the contents of the control tables generated by such automated procedures to demonstrate the correctness of the data. This can be carried out by proving theorems asserting the validity of the data in the control table. This process of theorem proving can be automated in a similar way as the network verifier described in Chapter 7.

Chapter 9

Interlockings and state machines

This chapter describes the dynamic states of track networks and a method of modelling interlockings using deterministic finite state machines.

9.1 States of a network

Recall from Chapter 5, the type :Tcix representing track circuits, the type :Point representing points and the type :Signal representing compound signals contain functions returning the states of the components. The network model of a track layout has these state functions embedded in it, thus it is a dynamic representation. At any given time t , the state of a network N , denoted by $S(N,t)$ is completely determined by the states of its constituent components, i.e., the values returned by their state functions. For example, the state of the network shown in Figure 6.4 is determined by the states of its six track circuits, six signals and two points.

For any given network N , there are three sets of state functions, denoted by

$tc(N)$ the set of track circuit state functions, $pnt(N)$ the set of point state functions and $sig(N)$ the set of signal state functions. The set $tc(N)$ can be written as the following expression

$$\text{IMAGE}(\text{TC.SFUNC} \circ \text{PART.CIRCUIT})(\text{VSN}) \quad (9.1)$$

This expression represents the image set obtained by applying the compound function $(\text{TC.SFUNC} \circ \text{PART.CIRCUIT})$ to the elements of the vertex set.¹ This function lifts the track circuit state function from each part. Each element of $tc(N)$ is a state function f_{tc_i} of the track circuit i . The type of the function is $\text{sum} \rightarrow \text{Tatate}$. Then, the state of the network at time t can be obtained by applying these functions to t . However, the results cannot be represented in sets. For example, a network N contains three track circuits, i.e., $tc(N) = \{f_{tc_1}, f_{tc_2}, f_{tc_3}\}$. Suppose that, at time t , the track circuit tc_1 is occupied and the others are clear. If the results of applying f_{tc_i} 's to t are stored in a set, it will be $\{\text{occupied}, \text{clear}\}$. This does not uniquely represent the state of the network. Therefore, a list has to be used. The state of the network N at t will be the list $[\text{occupied}; \text{clear}; \text{clear}]$.

An abbreviated type `:NetworkState` is defined to represent the state of a network. It is a compound of states of three kinds of dynamic components:

`:(Tatate)list # (PointState)list # (SignalState)list`

where the type `:PointState` is an abbreviation for the pair consisting of the position and locking state of points and the type `:SignalState` represents the state of a compound signal. Their definitions are listed in Table 9.1.

¹The image set of a function f on a set s , i.e., the expression $\text{IMAGE } f \ s$, is equal to the set $\{f \ x \mid x \in s\}$.

Abbreviation	Definition
<code>:NetworkState</code>	<code>:(Tatate)list # (PointState)list # (SignalState)list</code>
<code>:PointState</code>	<code>:Ppos # Ploc</code>
<code>:PointStateFunc</code>	<code>:(num -> Ppos) # (num -> Ploc)</code>
<code>:SignalState</code>	<code>:(MAspect # bool # SubAspect) + ShAspect</code>
<code>:SignalStateFunc</code>	<code>:(num -> MAspect) # (num -> bool) # (num -> SubAspect) + (num -> ShAspect)</code>

Table 9.1: Abbreviated types for states and state functions.

State functions Functions can be defined to lift the state functions from the respective components using expressions similar to 9.1. Three such functions are defined for obtaining lists of state functions of the three kinds of components. They are `TC.STATE.FUNCS`, `PNT.STATE.FUNCS` and `SIG.STATE.FUNCS`, and their definition are listed below.

HOL Definition 96 (TC.STATE.FUNCS_DEF)

```
"TC.STATE.FUNCS (N:Network) =  
  SET_LIST (IMAGE (TC_SFUNC o PART_CIRCUIT) (VS N))"
```

HOL Definition 97 (PNT.STATE.FUNCS_DEF)

```
"PNT.STATE.FUNCS (N:Network) =  
  let plat = SET_LIST (IMAGE PART_POINT (VS N)) in  
  (MAP (\p. (PNT_POS p, PNT_LOC p)) plat)"
```

HOL Definition 98 (SIG.STATE.FUNCS_DEF)

```
"SIG.STATE.FUNCS (N:Network) =  
  let siglat = SET_LIST (IMAGE (ELBL_SIGNAL o elb) (ES N)) in  
  (MAP SIG_SFUNC siglat)"
```

In the definition `PNT_STATE_FUNCS_DEF`, the local value `plst` is a list of points. The lambda expression $(\lambda p. (PNT_POS\ p, PNT_LOC\ p))$ is an anonymous function. When applying to a point, it extracts the position and locking state functions. Similarly, in the definition `SIG_STATE_FUNCS_DEF`, the local value `siglst` is a list of signals. The function `SIG_FUNC` returns the state function of the signal.

In all three functions, the polymorphic function `SET_LIST` is used to convert a set to a list. This set-to-list conversion function is characterized by the following theorem:

HOL Theorem 65 (`SET_LIST_THM`)

$\vdash \forall s. \text{FINITE } s \supset$

$(\forall x. (x \text{ IN } s = \text{ELEM}(\text{SET_LIST } s) x) \wedge (\text{CARD } s = \text{LENGTH}(\text{SET_LIST } s)))$

This theorem asserts that the function `SET_LIST` delivers a list containing all elements of a finite set and the length of this list is equal to the cardinal number of the set. The use of such conversion function in networks is justified because all networks are finite.

Instantaneous state Combining the above functions, a function delivering the instantaneous state of a network can be defined.

HOL Definition 99 (`NETWORK_STATE_DEF`)

```
"NETWORK_STATE (N:Network) t =
  let cflst = TC_STATE_FUNCS N in
  let plst = PNT_STATE_FUNCS N in
  let sflst = SIG_STATE_FUNCS N in
  ((APPLY (\(x t. f t) cflst t),
   (APPLY (\(f1 x2) t. (f1 t, x2 t)) plst t),
   (APPLY APPLY_SIG_FUNC sflst t)))"
```

Thus, the state of a network N at a given time t is specified by the expression `NETWORK.STATE N t` . The function `APPLY` used in `NETWORK.STATE` requires some explanation. Its definition is as below.

HOL Definition 100 (APPLY.DEF)

```
"(APPLY f □ x = (□ : (●●)list)) /\
  (APPLY (f:●●→(●→●●)) (CONS hd tl) (x:●) =
    CONS (f hd x) (APPLY f tl x))"
```

The purpose of this function is to provide a uniform conversion function for converting the dynamic structure—the structure of time varying functions—to the static structure representing the instantaneous state.

The second argument of `APPLY` is a list of dynamic structures $[g_1; \dots; g_n]$. For different kinds of components, their structures of state functions are different. The structure for track circuits is just a simple state function, i.e., each g_i is a track circuit state function. The structure for points is a pair, i.e., $g_i = (f_{pos}, f_{loc})$. The structure for signals is rather complicated, and the detail of this can be found in the ML source listed in Appendix F.

The last argument x of `APPLY` is the common argument to be supplied to the state functions in the structures. The first argument f is a higher-order function which applies the state functions in the structures g_i to x . This functional takes care of the difference in the structures.

By using appropriate functionals with `APPLY`, as seen in the `NETWORK.STATE.DEF`, the conversion from the dynamic structures of state functions to the static structures of instantaneous state can be written concisely and uniformly.

Example Using the naming convention described in Chapter 7, the passing loop network shown in Figure 6.4 has the following state function structure lists:

$\{[C1; C2; C3; C4; C5; C6], [N11; N12],$
 $[S100; S101; S102; S103; S104; S105]\}$

Suppose that, at time t , this network is in a state in which:

- all the track circuits are clear, i.e., $C_i t = \text{clear}$ for all i ;
- the point $N11$ is at its NORMAL position while $N12$ is at REVERSE and both are free to move, i.e., $(f_{posN11}^t, f_{locN11}^t) = (\text{normal}, \text{free_move})$ and $(f_{posN12}^t, f_{locN12}^t) = (\text{reverse}, \text{free_move})$;
- the entry signals to the network, namely $S100$ and $S101$, are OFF but all other signals are ON, i.e., $S_i t = (\text{green}, \text{ARB}, \text{ARB})$ for $i = 100$ or 101 and $S_i t = (\text{green}, \text{ARB}, \text{ARB})$ for other signals;²

where ARB indicates an arbitrary value of an appropriate type. Then, the state of the network is the triple below:

$([\text{clear}, \text{clear}, \text{clear}, \text{clear}, \text{clear}, \text{clear}],$
 $[(\text{normal}, \text{free_move}); (\text{reverse}, \text{free_move})],$
 $[(\text{green}, \text{ARB}, \text{ARB}); (\text{green}, \text{ARB}, \text{ARB}); (\text{red}, \text{ARB}, \text{ARB});$
 $(\text{red}, \text{ARB}, \text{ARB}); (\text{red}, \text{ARB}, \text{ARB}); (\text{red}, \text{ARB}, \text{ARB})])$

²Unlike the points, the structure of signal state functions has not been shown explicitly due to its complexity. The expression $S_i t$ is used to indicate that t is supplied as a common argument to the state functions in the structure S_i .

Dynamic network The topology of a network N at a given time t is based on the position of the points, i.e., the values returned by the list of point position functions. If a point p is at NORMAL position, effectively the reverse edges which connect the part containing p to its reverse successor are disconnected, because no movement along these edges can be made. Thus, the dynamic connectivity of a network, $D(N, t)$ is a graph obtained by deleting all the edges which represent the impossible movement to/from a point. The specification of $D(N, t)$ in HOL is the function DNETWORK.

HOL Definition 101 (DNETWORK_DEF)

```
"DNETWORK (N:Network) t = (VS B).
  {e | (e IS_EDGE B) /\
    (IS_PPART (e_src e) =>
      ((PNT_NORMAL (PART_POINT (e_src e)) t) /\
        (PART_ID (e_des e) = PART_PNT_NORMAL (e_des e)) \/\
        (PNT_REVERSE (PART_POINT (e_src e)) t) /\
        (PART_ID (e_des e) = PART_PNT_REVERSE (e_src e)) \/\
        (PART_ID (e_des e) = PART_PNT_TRAILING (e_src e))) |
      (IS_PPART (e_des e) =>
        ((PNT_NORMAL (PART_POINT (e_des e)) t) /\
          (PART_ID (e_src e) = PART_PNT_NORMAL (e_des e)) \/\
          (PNT_REVERSE (PART_POINT (e_des e)) t) /\
          (PART_ID (e_src e) = PART_PNT_REVERSE (e_des e)) \/\
          (PART_ID (e_src e) = PART_PNT_TRAILING (e_des e))) |
        T)) }"
```

The $D(N, t)$ of the passing loop network at the time t described above is as shown in Figure 9.1. Obviously, $D(N, t)$ is a subgraph of the underlying network N , but it may not still be a network.

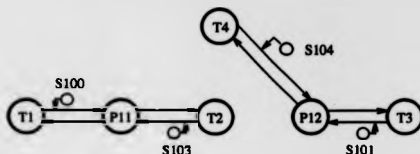


Figure 9.1: A graph representing the passing loop at t .

9.2 Proving routes

One typical task of an interlocking is to set up or prove a route. A route is proved if the required functions as specified in the control table are all satisfied. Recall from Chapter 8, that these required functions are specified by a set of functions which corresponds to the safety rules. These functions are:

TCIRCUITS for the required track circuits,

NORM.POINTS for the points required NORMAL,

REV.POINTS for the points required REVERSE,

EXIT.SIGNAL for the exit signal,

ENTRY.SIGNALS for the entry signals of the conflicting routes.

CR.TCIRCUITS for the track circuits along the conflicting routes.

The specification for a route being *proved* will be a predicate asserting that the components returned by the above functions are at the required states. The HOL function `ROUTE_PROVED` is defined for this purpose.

HOL Definition 102 (ROUTE_PROVED_DEF)

```

"ROUTE_PROVED r1 x t =
  let r1st = CONFLICT_ROUTES r1 x in
  ((EVERY (\x. TC_CLEAR x t) (TCIRCUITS x)) /\
   (EVERY (\p. PST_NORMAL p t) (NORM_POINTS x)) /\
   (EVERY (\p. PST_REVERSE p t) (REV_POINTS x)) /\
   (EVERY (\s. ~(SIGNAL_FAULT s t)) (EXIT_SIGNAL x)) /\
   (EVERY (\s. ON s t) (ENTRY_SIGNALS x r1st)) /\
   (EVERY (\x. TC_CLEAR x t) (CB_TCIRCUITS x r1st)))"

```

In the above definition, the first argument $r1$ is the list of all routes in the network. It is required to work out the conflicting routes. **EVERY** is a pre-defined constant in HOL. The expression **EVERY** $P[x_1; \dots; x_n]$ evaluates to T if and only if Px_i are true for $1 \leq i \leq n$.

The action of setting up a route is usually initiated by the signalman, or by another system, such as the ARS system mentioned in Section 3.3.3, sending a request to the interlocking. The interlocking then checks the states of the required functions, and if any required points are not in the right position or if the required signals do not display the correct aspects, it will attempt to change their positions or aspects to the required states. If this succeeds the route is proved, and the predicate **ROUTE_PROVED** returns true. This procedure of route setting can be modelled as a state machine which is discussed in the next section.

9.3 Interlockings

Based on the discussion in the previous two sections, interlocking systems can be modelled by a finite deterministic automaton or finite state machine. The formalization of state machines in HOL is described first, then a method of using this formal theory in the modelling of interlockings is described.

9.3.1 State machine theories

Theories of deterministic and non-deterministic finite state machines in HOL have been developed by Loewenstein[47][46]. In his theory LSA, a labelled state automaton is represented by the predicates LSA or PLSA.

HOL Definition 103 (LSA)

"LSA (Q,N) = ?s: num \rightarrow *. Q (s 0:*, s 0) \wedge
 (!t. N (s t, s t) (s (SUC t), s(SUC t)))"

HOL Definition 104 (PLSA)

"!(Q:(***) \rightarrow bool) P N s. PLSA (Q,P,N) s =
 (?s. Q(s 0, s 0) \wedge
 (!t. P(s t, s t) \wedge (!t. N(s t, s t)(s(SUC t), s(SUC t)))))"

In the definition of LSA, Q is a predicate asserting a set of possible initial states, N is a predicate asserting a set of next states, and e is the external signals including all the inputs and outputs. LSA specifies that there exists a state function s such that $s(0)$ is an initial state and, given any time t , the state transition from $s(t)$ to $s(t+1)$ satisfies the next state function. The definition of PLSA is similar except that a predicate P asserting some properties of the machine is explicitly stated.

This theory can be used in reasoning about properties of a given machine based on the theorem LSA.eq.PLSA.

HOL Theorem 66 (LSA.eq.PLSA)

$\vdash \forall QPN.$

$(\forall e.s. Q(e,s) \supset P(e,s)) \wedge (\forall e.s.e'. N(e,s)(e',s') \wedge P(e,s) \supset P(e',s')) \supset$

$(\forall e''. \text{LSA}(Q,N)e'' = \text{PLSA}(Q,P,N)e'')$

This theorem states that, if the machine has property P at its initial state, and if the machine has property P at a state s implies the property also holds at the next state s' , then the machine has the property P . This theory can also be used in verifying the implementation of a machine by virtue of the theorem $LSA \text{ imp } LSA$.

HOL Theorem 67 ($LSA \text{ imp } LSA$)

$$\begin{aligned} & \vdash \forall Q_1 Q_2 N_1 N_2. \\ & (\exists R. (\forall e s_1. Q_1(e, s_1) \supset (\exists s_2. Q_2(e, s_2) \wedge R e s_1 s_2)) \wedge \\ & (\forall e' s_1 s'_1 s_2. R e s_1 s_2 \wedge N_1(e, s_1)(e', s'_1) \supset \\ & (\exists s'_2. R e' s'_1 s'_2 \wedge N_2(e, s_2)(e', s'_2)))) \supset \\ & (\forall e. LSA(Q_1, N_1) e \supset LSA(Q_2, N_2) e) \end{aligned}$$

This states that, if there exists a mapping R between the states of the machine specified by (Q_1, N_1) and the machine specified by (Q_2, N_2) , then the two machines are equivalent.

A pilot study of using this theory in the modelling of a level crossing protection has been carried out[26]. In this study, the control system of the barrier and signals was specified as an LSA machine with initial states `INIT` and next state function `NEXT`. The major safety property of this machine was specified such that not both railway traffic and road traffic is allowed to proceed at the same time. A theorem stating that the crossing control machine has such safety property was derived. Some details of the definitions and theorems mentioned are listed in Appendix E.



Figure 9.2: An interlocking state machine.

9.3.2 Interlocking as state machine

Using the theory described in above, an interlocking can be modelled by a deterministic finite state machine LSA. Shown in Figure 9.2 is a generic interlocking state machine consisting of two input streams, I_{req} and I_{sen} , a list of outputs, O_{act} , and a list of states S_{inter} .

The input stream I_{req} feeds requests to the interlocking to perform some actions, such as setting up a route. Suppose that the only possible requests are to set up or to clear a route. A request can then be represented by a pair $(action, route)$ where *action* is either *set* or *clear* and *route* is the name of the route, e.g. S100S102.

The input stream I_{sen} consists of the inputs from all the sensors of network components, such as track circuits and signal proving circuits. These reflect the current state of the network as described in Section 9.1.

The outputs in O_{act} drives the actuators, such as point machines and signals.

The LSA machine combines all the external signals into a single argument e . For the generic interlocking machine, e would be a triple $(O_{act}, I_{req}, I_{sen})$.

The states S_{inter} is the current internal states of the interlocking. The generic

interlocking state machine may have the following internal states:

- *init* is the initial state, in which no routes have been set up, all track circuits are CLEAR, all points are at their NORMAL position and all signals are ON;
- *all_clear* is the state in which no routes have been set up;
- *proving* is the state the interlocking performs the tasks of proving a route;
- *clearing* is the state the interlocking performs the tasks of clearing a route;
- *route_set* is the state in which at least one route has been set up.

As part of the internal state, a set *routeset* containing all the routes which have been set up should be included in S_{inter} . Clearly, in the state *all_clear*, *routeset* must be empty. Thus, the state function *s* in the LSA machine should return a pair (*state*, *routeset*). The state transition diagram for this generic interlocking machine is shown in Figure 9.3.

Based on this state transition diagram and the above discussion, the initial state predicate can be expressed in HOL as the definition *INIT_DEF*.

HOL Definition 105 (INIT_DEF)

```
"INIT ((O_act, I_req, I_sen), S) =
  (S = (init, { }))
  (EVERY (\t. t = clear) (FST I_sen)) /\
  (EVERY (\(p1,p2). p1 = normal) (FST(SND I_sen))) /\
  (EVERY SIG_OW (SND(SND I_sen)))"
```

Similarly, the next state function can be specified as function *NEXT* shown below:

```
"NEXT ((O_act, I_req, I_sen), (state, routeset))
  ((O_act', I_req', I_sen'), (state', routeset')) =
  ((state = init) /\ (state = all_clear)) =>
  (((FST I_req) = set) => (state' = proving) | (state' = state)) |
  (state = proving) =>
  (PROVE_ROUTE routeset (SND I_req) /\ (state' = route_set) /\
```

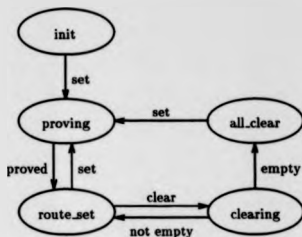


Figure 9.3: A state transition diagram of generic interlocking machine.

```

(routeset' = ((SEND I_req) INSERT routeset)) |
(state = route_set) =>
(((FST I_req) = set) => (state' = proving) | (state' = clearing)) |
% state = clearing %
(((SEND I_req) IN routeset) =>
  (?x. (routeset = {x}) =>
    (CLEAR_ROUTE routeset (SEND I_req)) /\ (state' = all_clear) /\
(routeset' = {}) |
    (CLEAR_ROUTE routeset (SEND I_req)) /\ (state' = route_set)) /\
  (routeset' = (routeset DELETE (SEND I_req))) |
  (state' = state) /\ (routeset' = routeset))
  
```

In this definition, the functions **PROVE_ROUTE** and **CLEAR_ROUTE** are called to perform the tasks of proving and clearing a route. Their specifications depend on:

1. the topology of the network, i.e., how complex is the network, how many routes are there and so on;
2. the safety rules, i.e., what are the required functions for each route;
3. the control algorithms, i.e., what are the possible requests and how to achieve them.

PROVE.ROUTE can use the predicate ROUTE.PROVED described in Section 9.2 and the functions for working out required track components described in Chapter 8 to check the conditions for setting up routes. The actions taken to prove a route will be to move and lock the points at the required position, to mark all the track circuit to be hooked to prevent them from being included in a conflicting route and to instruct the signals to display the correct aspect. Note that PROVE.ROUTE proves a route in the context of *routeact*, which is passed as an argument, so that the route cannot be proved if it is in conflict with any one of the route already set up.

Now, properties of the generic interlocking machine can then be derived using the method described in Section 9.3.1., i.e., by defining predicates expressing the required properties and proving theorems asserting the equivalence of LSA and PLSA machines. The most important properties of an interlocking are *safety* and *liveness*.

One of the desired safety properties that any interlocking should possess is that no conflicting routes can be set up at any time. This statement can be expressed in HOL as

```

IF x1 ≠ x2. (ROUTE # x1) ∧ (ROUTE # x2) ∧ ¬(r1 = r2) ∧
(CONFLICTING_ROUTE # x1 x2) ==>
!v. ¬((PROVED_ROUTE # x1 v) ∧ (PROVE_ROUTE # x2 v))

```

A predicate NO.CONFLICTING.ROUTES can be define to have this property. To show the generic interlocking machine has this property, the following goal can be set up:

```

!a. LSA(INIT, NEXT) a = PLSA(INIT, NO_CONFLICTING_ROUTES, NEXT) a

```

Induction can be used to solve this goal. The induction principle for the deterministic state machine has been encoded in the theorem LSA_eq_PLSA. *Modus ponens* can be used with this theorem to solve the goal if the following two subgoals can be solved:

1. the base case:

$$\text{to } s. \text{ INIT } (s, s) \implies \text{NO_CONFLICTING_ROUTES } (s, s)$$

2. the induction step:

$$\text{to } s \neq s'. \text{ NEXT}(s, s) \wedge \text{NO_CONFLICTING_ROUTES } (s, s) \implies \\ \text{NO_CONFLICTING_ROUTES } (s', s')$$

The base case is clearly true because no route is proved in the initial state. The induction step is also true because the route proving functions called in NEXT require all the track circuits along the route to be CLEAR. Other desired properties can be deduced in the same way. Implementation of this generic interlocking machine can then be specified, and verification can be carried out by proving theorems of the form

$$\text{to. LSA}(\text{INIT_IMP}, \text{NEXT_IMP}) s \implies \text{LSA}(\text{INIT}, \text{NEXT}) s$$

where INIT_IMP and NEXT_IMP are the initial and next state functions of the implementation.

The method of modelling interlocking using state machine has been described. This generic interlocking state machine is only the top-level. A large amount of research effort is still required to refine this model and to develop practical implementation.

Chapter 10

Conclusions and future research

To conclude this dissertation, a general discussion of the creation of a generic abstract model, the applications of such a model, the suitability of HOL and issues in the method of the research is given in this chapter. Possible future researches are indicated.

10.1 A generic abstract model

The **NETWORK** theory is a generic abstract model of railway track networks based on well-founded mathematics. This model is abstract because it only captures the essential topological relations of the network components and disregards any physical constraints and implementation technology. The model is primarily used for writing top-level specifications of interlocking systems and reasoning about their logical operations. The networks being modelled are assumed to satisfy all physical constraints specified by regulations of the railway authorities, for example the distance between successive signals is not less than the service braking distance.

Because of this abstraction, the theory is general enough to be used to model railway track networks of railway authorities with different standards. Although the terminology, interpretations, regulations and implementations may vary between different railway authorities, the basic principles of fixed block interlockings are essentially the same. These are:

- division of tracks into block sections;
- detection of trains by track circuit or other means;
- regulation of traffic by signal aspects.

All these principles are expressed and modelled in the theory. The sectioning of the track leads to track component parts and the representation of parts by vertices of graphs. The notion of a train has not been explicitly modelled, but by the occupied state of track circuits. The modelling of signal has contained a fair amount of detail, but the importance of the ON and OFF aspects has been stressed.

The theory has been developed in a rigorous manner, i.e., using only definitional extensions without introducing new axioms. It is based on the set theory and graph theory within the framework of HOL logic, therefore it is consistent. To improve the readability of the theories, the names of the logical constants are generally quite long.

10.2 Applications of the model

The safety record of the railway industry has been very good. This is achieved by the rigorous regulations developed through decades of working experience. As more

and more new technology is being adopted by the industry, especially the use of microprocessors in controlling vital safety functions, it is very important to ensure that the high integrity and reliability of the interlocking systems are maintained. Due to the complexity of the new technology, more rigorous methods should be employed in the analysis, development and implementation of signalling systems.

Three case studies of applying the NETWORK theory have been described in Part III. The first two are in the area of signalling scheme design. The model is used in the CAD tools which help the signalling engineers in designing interlocking systems. This is certainly not the most critical part of an interlocking system. The reason for applying the network model in this area first is because the theory is new, and even the whole approach to signalling system design using formal methods is itself new. It would not be wise to apply it in the most critical functions, such as the hardware and software of the interlocking processors, before more experience has been gained. This does not mean that there is lack of confidence on the abstract modelling of the track networks, but just to be more cautious in introducing new technology in safety-critical systems so that to minimize the risk of introducing unsafe factors into the systems.

Nevertheless, the generation and verification of formal specifications of railway track layout, and creation of control tables are very important steps in the process of signalling scheme design. By using more rigorous methods, more mistakes can be discovered in earlier stage, thus, it leads to better design and reduces costs.

Chapter 9 indicates a possible method of modelling, specifying and reasoning the vital safety functions of interlocking systems using the dynamic state of the network model and finite state machines. There are still many difficult issues which

need to be solved before practical systems can be implemented, thus, more research is required. The example interlocking state machine shown in that chapter may have been oversimplified because many important features have been ignored, such as approach lockings. The actions for setting up and clearing routes have not been completely specified. These have to be solved in order to model and to formally specify a real interlocking system.

Safety is the paramount requirement of any interlocking system. Ultimately, an interlocking system is safe if it never allows any traffic which may lead to potential collision, even if there is equipment failure. The goal, stating that no conflicting routes can be proved at the same time, shown in Section 9.3.2 is only one of many properties a safe interlocking system should possess. Examples of propositions contributing to the total safety of a interlocking system include:

- a route cannot be set up until all required functions are satisfied;
- an approach locked point cannot be switched.

The ultimate goal of liveness of a railway network can be expressed as the ability of running the timetable. Some of provable liveness statements are:

- there is a route from A to B for some specific A and B;
- there exists a time, the required route from A to B can be set up.

When using a state machine to model the control loop of interlockings, the eventual occurrence of the transition to the next state and the maximum time required for this transition to happen are two key properties contributing to the total safety and liveness of the system.

Traditionally, the concept of liveness is not treated as being important and explicit as safety. This is probably because there is always provision for overriding the system, such as giving authority to a driver to pass a signal displaying the ON aspect in situations arisen from faulty equipment. This prompts a theory of abnormal operations, i.e., a theory modelling the regulations and procedures for recovery from equipment failure.

The properties discussed in the previous paragraphs imply that a full temporal logic may be required in the modelling of the interlocking. Furthermore, the generic interlocking state machine specified in Section 9.3.2 is only a top-level specification. To develop a practical interlocking system still requires a large amount of research effort to refine this top-level model down to some detail levels before an implementation can be developed. The verifications between these levels have to be carried out to show the implementation model possesses the same safety properties.

10.3 The HOL system

The application of formal methods in the design of railway signalling systems is still a new subject. From the experience of the current research carried out by the author, HOL has the following four major advantages: generality, conciseness, consistence and extensibility.

Generality is due to the underlying logic which is a general higher-order mathematical logic. Although, HOL was initially used for hardware verification, it has not been designed for any specific application, its generality making it well suited for many diverse applications areas. It is certainly suitable for railway signalling

application. Because of its generality, one can deduce general properties for all networks.

Conciseness is due to the fact that the underlying logic is higher-order. This allows very compact expressions being written. In addition, higher-order functions provides a very suitable way of modelling time-varying states of the system and components.

Consistence is guaranteed by the strongly typed meta-language and the ways a theory can be extended. By using only definitional extensions to theories, no logical contradiction will be introduced.

Extensibility provides a powerful means of adapting the system for different application areas. Since the user is able to add new libraries and new theories, a sub-system with interface more convenient for the application at hand can be developed.

The HOL system is an integrated system for developing system specifications at various levels and for reasoning and verifying them. However, there are some drawbacks. The learning curve is very steep, and demands considerable skill if the theorem prover is used efficiently. This hinders its use by practicing engineers and designers who are usually not familiar with formal logic and theorem proving. This can be remedied by providing better interface to the system, such as a windowing environment, and by providing more automatic tools for specific applications, such as the network verifier.

10.4 Methodology issues

The approach of this research is to develop a general theory and then apply it to more specific problem. The advantage of this approach is that the general theory, namely the graph theory, provides a sound mathematical structure as a foundation on which more specific arguments can be based. Railway track networks are fairly complicated, an abstract model of them must rely on some well-founded structures. They are intrinsically well suited to be represented by graphs. Adopting graphs as the structure of track networks allows many well known algorithms on graphs to be used; particularly important are the algorithms for finding paths. Without a properly defined data structure, it would be very difficult to specify and model track networks and to derive their useful properties.

Graph theory has been applied in numerous practical problems in very diverse scientific and engineering fields. In addition to its use in railway signalling, the graph theory developed in HOL can provide a starting point for other applications. For example in the transport industry, the problem of finding the most economical route of delivering goods and the problem of maximizing the network capacity can be solved using graphs.

Verification by automatic theorem proving is a method developed in this research for verifying railway layout designs (described in Chapter 7). This approach to verifications relies on the development of a generic abstract model, and a proof strategy based on this model. Each specific design is then verified against this generic model, and specific instances of the general theorem are automatically deduced using the proof strategy. This approach can be used in other application areas as well.

The major advantages of this approach are that it automates the theorem proving process and it provides an easy-to-use tool for practicing engineers and designers. Since there is a misconception that formal methods are very difficult to use, the second point is very important in persuading industry to adopt them.

The research presented in this dissertation is a small step towards the applications of formal methods in practical safety-critical system design. The author hopes that this will spark off more research effort in applying formal methods in system analysis and design in the signalling industry and other industries which use safety-critical systems.

Bibliography

- [1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.
- [2] K. Akita, T. Watanabe, and H. Nakamura. Solid-state interlocking in railway signalling, SMILE. In *Proceedings of the International Conference on Electric Railway Systems for a New Century*, pages 294-298. IEE, September 1987.
- [3] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth through Proof*. Computer Science and Applied Mathematics Series. Academic Press, 1986.
- [4] W. R. Bevier. Kit and the short stack. *Journal of Automated Reasoning*, 5(4), November 1989.
- [5] W. R. Bevier, W. A. Hunt, J. S. Moore, and W.D.Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4), November 1989.
- [6] D. Bjørner and Cliff B. Jones. *Formal Specification and Software Development*. Prentice-Hall International, 1982.
- [7] D. Bjørner, C.A.R.Hoare, and H. Langmaack, editors. *VDM '90, VDM and Z — Formal Methods in Software Development*. Lecture Notes in Computer Science, No. 428. Springer-Verlag, 1990.

- [8] Robert S. Boyer and J Strother Moore. *A Computational Logic*. Perspectives in Computing. Academic Press, Inc., San Diego, CA, U.S.A, 1979.
- [9] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Perspectives in Computing. Academic Press, Inc., San Diego, CA, U.S.A, 1988.
- [10] Robert S. Boyer and J Strother Moore. A theorem prover for a computational logic. In M. E. Stickel, editor, *Proceedings of 10th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, pages 1-15, Kaiserslautern, FRG, July 1990. Springer-Verlag.
- [11] B. A. Carré. *SPADE Static Code Analysis Manual*. Program verification Ltd., April 1985.
- [12] B. A. Carré and T. J. Jennings. SPARK—the SPADE Ada kernel. Technical report, University of Southampton, 1988.
- [13] K. Celinski. Microcomputer controllers introduce modern technology in fail-safe signalling. In *Proceedings of the International Conference on Electric Railway Systems for a New Century*, pages 310-314. IEE, September 1987.
- [14] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56-68, 1940.
- [15] A. Cohn. A proof of correctness of the viper microprocessor: the first level. Technical report, University of Cambridge Computer Laboratory, 1988.
- [16] A. Cohn. A proof of correctness of the viper microprocessor: the second level. Technical report, University of Cambridge Computer Laboratory, 1990.

- [17] Computer Laboratory, University of Cambridge. *The HOL System : Description*, 1990.
- [18] Computer Laboratory, University of Cambridge. *The HOL System : Tutorial*, 1990.
- [19] D. Craigen and K. Summerskill, editors. *Formal Methods for Trustworthy Computer Systems(FM89)*, Workshops in Computing. Springer-Verlag, 1990.
- [20] Alan Cribbens. The solid state interlocking. In *Proceedings of the International Conference on Railway Safety, Control and Automation towards the 21st Century*, pages 24 - 29, Sept. 1984.
- [21] Alan Cribbens. A solid state interlocking (ssi): an integrated electronic signalling system for mainline railways. In *IEE Proceedings, Part B*, volume 134, pages 148 - 158, MAY 1987.
- [22] Alan H. Cribbens, M. J. Furniss, and H. A. Ryland. The solid state interlocking project. In *Proceedings of the International Conference on Railways in the Electronic Age*, pages 1 - 5, Nov. 1981.
- [23] W. J. Cullyer. *Implementing Safety Critical Systems: The VIPER microprocessor*, pages 1-26. Kluwer Academic Publishers, 1987.
- [24] W. J. Cullyer. Safety-critical control systems. *Computing & Control Engineering Journal*, 2(5):202-210, September 1991.
- [25] W. J. Cullyer and Wong W. Application of formal methods to railway signalling—a case study. *IEE Computer and Control Engineering journal*, 1992. Submitted to IEE CCEJ.

- [26] W. J. Cullyer and W. Wong. A mathematical approach to the protection of grade crossing. In *Proceedings of international symposium on rail-highway grade crossing research and safety*, Knoxville, Tennessee, USA, 31st Oct - 3rd Nov 1990.
- [27] Norman Delisle and David Garlan. A formal specification of an oscilloscope. *IEEE Software*, September 1990.
- [28] Antoni Diller. *Z—An Introduction to Formal Methods*. Jonh Wiley & Sons, 1990.
- [29] Alan Gibbon. *Algorithmic Graph Theory*. Cambridge University press, Cambridge England, 1985.
- [30] J. Goguen, C. Kirchner, H. Kirchner, A. Megrelis, J. Meseguer, and T. Winkler. An introduction to OBJ3. In S. Kaplan and J. P. Jouannaud, editors, *Conditional Term Rewriting Systems. — 1st International workshop proceedings*. Springer-Verlag, 1988.
- [31] J. A. Goguen. OBJ as a theorem prover with applications to hardware verification. In G. Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, chapter 5, pages 218-267. Springer-Verlag, 1989.
- [32] J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R. T. Yeh, editor, *Current Trends in programming Methodology, Vol. IV — Data Structuring*. Prentice-Hall, 1977.

- [33] Michael. C. Gordon. Mechanizing programming logics in higher order logic. In *Current Trends in Hardware Verification and Automated Theorem Proving*, chapter 10, pages 387-439. Springer-Verlag, 1989.
- [34] Michael J. Gordon. *HOL: A Proof Generating System for Higher-Order Logic*, pages 73-128. Kluwer Academic Publishers, 1987.
- [35] Michael J. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF*. Lecture Notes in Computer Science, No. 78. Springer-Verlag, 1979.
- [36] F. K. Hanna and N. Daeche. Specification and verification using higher-order logic: A case study. In G. J. Milne and P. A. Subrahmanyam, editors, *Formal Aspects of VLSI Design*, pages 179-213. Springer-Verlag, 1986.
- [37] Health and Safety Executive. *Guidance on the Use of Programmable Electronic Systems in Safety-related Applications*, 1986.
- [38] W. A. Hunt. Microprocessor design verification. *Journal of Automated Reasoning*, 5(4), November 1989.
- [39] Warren A. Hunt. *FM8501: A Verified Microprocessor*. PhD thesis, The University of Texas at Austin, 1985.
- [40] IEC. *Functional Safety of Programmable Electronic Systems*. IEC SC65A/WG10 3rd Draft, June 1989.
- [41] IEC. *Software for Computers in the Application of Industrial Safety Related Software*. IEC SC65A/WG9 3rd Draft, June 1989.
- [42] Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall, London, 1986.

- [43] J. J. Joyce. Formal specification and verification of asynchronous processes in higher-order logic. In *Specification and Verification of Concurrent Systems — Proceedings of BCS-FACS Workshop (TR45)*, 1988.
- [44] J. J. Joyce. Totally verified systems: Linking verified software to verified hardware. Technical report, University of Cambridge Computer Laboratory, 1989.
- [45] J.S.Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4), November 1989.
- [46] P. Loewenstein. Formal verification of state-machines using higher-order logic. In *Proceedings of 1989 IEEE International Conference on Computer Design: VLSI in computers and processors*. IEEE, IEEE Computer Society Press, 1989.
- [47] P. Loewenstein. Reasoning about state machines in higher-order logic. In M. Leiser and G. Brown, editors, *Hardware Specification, Verification and Synthesis: Mathematical Aspect*, Lecture Notes in Computer Science No. 408, pages 67-89. Springer-Verlag, 1989.
- [48] T. F. Melham. Automating recursive type definition in higher-order logic. In Graham Birtwistle and P. A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341-386. Springer-Verlag, 1989.
- [49] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. The MIT Press, 1990.
- [50] I. H. Mitchell. The design and testing of application database for a railway signalling system. In *Proceedings of International Conference on Software Engineering for Real-time Systems*, pages 159 - 164, September 1987.

- [51] MOD. *Requirements for Hazard Analysis of Safety-related Computer Systems*. Draft UK DefStan 00-56, April 1991.
- [52] MOD. *Requirements for the procurement of safety-critical software in defence equipment*. Draft UK DefStan 00-55, April 1991.
- [53] J. D. Murchland. A new method for finding all elementary paths in a complete directed graph. Technical Report LSE-TNT-22, London School of Economics, 1965.
- [54] O. S. Nock, editor. *Railway Signalling: A treatise on the recent practice of British Railways*. A and C Black, London, 1980.
- [55] L. C. Paulson. *ML for the Working Programmer*. Cambridge university press, 1991.
- [56] W. L. Price. *Graphs and Networks: An Introduction*. Butterworth Co (Publishers) Ltd., London, 1971.
- [57] RCTS. *Software Considerations in Airborne Systems and Equipment Certification*. RCTS-178A, May 1985.
- [58] J. Rushby. Formal methods and critical systems in the real world. In D. Craigen and K. Summerskill, editors, *Formal Methods for Trustworthy Computer Systems (FM89)*, Workshops in Computing, pages 121-125. Springer-Verlag, 1990.
- [59] Roger Shaw and Cliff B. Jones. *Case Studies in System Software Development*. Prentice-Hall International series in Computer Science. Prentice-Hall International, September 1989.
- [60] R. C. Short. Software validation for railway signalling and train control systems.

- In *Proceedings of the International Conference on Electric Railway Systems for a New Century*, pages 315-319. IEE, September 1987.
- [61] Frances Singer. IECC — a new era in british rail signalling. *Modern Railways*, pages 533-535, October 1989.
- [62] J. M. Spivey. *The Z Notation — A Reference Manual*. Prentice Hall International Series in Computer Science. Prentice-Hall, 1989.
- [63] B. A. Sufrin. Formal specification of a display editor. *Science of Computer Programming*, 1:157-202, 1982.
- [64] Open University. *Graphs, Networks and Design— Unit 2: Graphs and Digraphs*. Open University press, 1981.
- [65] Open University. *Graphs, Networks and Design— Unit 5: Paths and Cycles*. Open University press, 1981.
- [66] W.D.Young. A mechanically verified code generator. *Journal of Automated Reasoning*, 5(4), November 1989.
- [67] W. Wong. Formatting hol text: the library latex-hol. HOL system library Manual, May 1991.
- [68] W. Wong. A simple graph theory and its application in railway signalling. In *Proceedings of the 1991 international Workshop on Higher Order Logic Theorem Proving System and Its Application*. IEEE, 1991.

Appendix A

HOL theories

This appendix lists all the theories described in Part II in the format similar to the output of the HOL utility function `print.theory`. Each theory is listed in a separate section. There are possibly six subsections in each theory: *parents* lists the parent theories, *types* lists the names of types defined in the current theory, *constants* lists the names and types of all constants defined in the current theory, *infix* lists the name of those constants which have infix syntactic status, *definitions* lists all theorems associated with constant definitions and *theorems* lists all theorems saved in the current theory.

A.1 The theory func

Parents

HOL sets

Constants

```
-->      ":(a)set -> ((ee)set -> ((e -> ee) -> bool))"
-->>     ":(e)set -> ((ee)set -> ((e -> ee) -> bool))"
>-->     ":(a)set -> ((ee)set -> ((e -> ee) -> bool))"
<-->     ":(e)set -> ((ee)set -> ((e -> ee) -> bool))"
FUN_INV  ":(a)set -> ((ee)set -> ((e -> ee) -> (ee -> a)))"
FUN_PINVERSE ":(e)set # (ee)set ->
          ((e -> ee) -> ((ee -> e) -> bool))"
```

```

FUN_INVERSE ":(*)set @ (**)set ->
  ((e -> ee) -> ((ee -> e) -> bool))"

```

Infixes

```

--> ":(*)set -> ((**)set -> ((e -> ee) -> bool))"
-->> ":(*)set -> ((**)set -> ((e -> ee) -> bool))"
>--> ":(*)set -> ((**)set -> ((e -> ee) -> bool))"
<--> ":(*)set -> ((**)set -> ((e -> ee) -> bool))"

```

Definitions

```

FUN_DEF      ⊢ VA B f. $--> AB f = (∀z. z IN A ⊃ f z IN B)
FUN_ONT0_DEF ⊢ VA B f. $-->> AB f = (∀z. z IN A ⊃ f z IN B) ∧
  (∀y. y IN B ⊃ (∃x. x IN A ∧ (y = f x)))
FUN_ONE_ONE_DEF ⊢ VA B f. $>--> AB f = (∀z. z IN A ⊃ f z IN B) ∧
  (∀x y. z IN A ∧ y IN A ∧ (f x = f y) ⊃ (x = y))
FUN_ISO_DEF  ⊢ VA B f. $<--> AB f = $>--> AB f ∧ $>-->> AB f
FUN_INV_DEF  ⊢ VA B f y. FUN_INV AB f y = ((y IN B ∧ (∃x. x IN A ∧
  (y = f x))) ⇒ (ex. x IN A ∧ (y = f x))) | (ex. x IN A)
FUN_PINVERSE_DEF ⊢ VA B f g. FUN_PINVERSE(A, B) f g = $--> AB f ∧
  $-->> B A g ∧ (∀z. x IN A ⊃ ($g f z = z))
FUN_INVERSE_DEF
  ⊢ VA B f g. FUN_INVERSE(A, B) f g = FUN_PINVERSE(A, B) f g ∧
  FUN_PINVERSE(B, A) g f

```

Theorems

```

FUN_ONT0_0 ⊢ VA B C f g. $-->> AB f ∧ $>-->> B C g ⊃ $-->> AC(g ∘ f)
FUN_ONE_ONE_0 ⊢ VA B C f g. $>-->> AB f ∧ $>-->> B C g ⊃ $>-->> AC(g ∘ f)
FUN_ISO_0 ⊢ VA B C f g. $<--> AB f ∧ $<--> B C g ⊃ $<--> AC(g ∘ f)
FUN_TY ⊢ VA B f. $>--> AB f ∨ $>--> AB f ⊃ $>--> AB f
FUN_INV_TY ⊢ VA B f. ¬(A = { }) ⊃ $--> B A (FUN_INV AB f)
LEFT_FINV ⊢ VA B f. ¬(A = { }) ∧
  $>--> AB f ⊃ FUN_PINVERSE(A, B) f (FUN_INV AB f)
RIGHT_FINV ⊢ VA B f. ¬(A = { }) ∧
  $>--> AB f ⊃ FUN_PINVERSE(B, A) (FUN_INV AB f) f

```

$\text{LEFT_RIGHT_PINV} \vdash \forall A B f g. \$ \rightarrow \rightarrow A B f \wedge \$ \rightarrow \rightarrow B A g \wedge$
 $\text{FUN_PINVERSE}(A, B) f g \supset \$ \rightarrow \rightarrow A B f \wedge \$ \rightarrow \rightarrow B A g$
 $\text{ISO_INVERSE} \vdash \forall A B f g. \$ \rightarrow \rightarrow A B f \wedge \$ \rightarrow \rightarrow B A g \wedge$
 $\text{FUN_INVERSE}(A, B) f g \supset \$ \rightarrow \rightarrow A B f \wedge \$ \rightarrow \rightarrow B A g$
 $\text{FUN_EMPTY_LEFT} \vdash (\forall B f. \$ \rightarrow \rightarrow \{ \} B f) \wedge (\forall B f. \$ \rightarrow \rightarrow \{ \} B f) \wedge$
 $(\forall B f. \$ \rightarrow \rightarrow \{ \} B f = (B = \{ \})) \wedge (\forall B f. \$ \rightarrow \rightarrow \{ \} B f = (B = \{ \}))$
 $\text{FUN_EMPTY_RIGHT} \vdash (\forall A f. \$ \rightarrow \rightarrow A \{ \} f = (A = \{ \})) \wedge$
 $(\forall A f. \$ \rightarrow \rightarrow A \{ \} f = (A = \{ \})) \wedge$
 $(\forall A f. \$ \rightarrow \rightarrow A \{ \} f = (A = \{ \})) \wedge (\forall B f. \$ \rightarrow \rightarrow A \{ \} f = (A = \{ \}))$
 $\text{FUN_I} \vdash (\forall A. \$ \rightarrow \rightarrow A A) \wedge (\forall A. \$ \rightarrow \rightarrow A A) \wedge (\forall A. \$ \rightarrow \rightarrow A A) \wedge$
 $(\forall A. \$ \rightarrow \rightarrow A A)$
 $\text{ISO_FINV} \vdash \forall A B f. \$ \rightarrow \rightarrow A B f \supset \$ \rightarrow \rightarrow B A (\text{FUN_INV } A B f)$

End of theory func

A.2 The theory graph

Parents

HOL sets func

Constants

$\text{IS_EDGE} \text{ " : * \# (\# \# \#) \rightarrow ((\#) \text{set } \# (\# \# (\# \# \#)) \text{set} \rightarrow \text{bool}) "}$
 $\text{IS_VERTEX} \text{ " : * \# \rightarrow ((\#) \text{set } \# (\# \# (\# \# \#)) \text{set} \rightarrow \text{bool}) "}$
 $\text{DELETE_EDGE} \text{ " : (\#) \text{set } \# (\# \# (\# \# \#)) \text{set} \rightarrow$
 $(\# \# (\# \# \#) \rightarrow (\#) \text{set } \# (\# \# (\# \# \#)) \text{set}) "}$
 $\text{DELETE_VERTEX} \text{ " : (\#) \text{set } \# (\# \# (\# \# \#)) \text{set} \rightarrow$
 $(\# \rightarrow (\#) \text{set } \# (\# \# (\# \# \#)) \text{set}) "}$
 $\text{INSERT_VERTEX} \text{ " : * \# \rightarrow ((\#) \text{set } \# (\# \# (\# \# \#)) \text{set} \rightarrow$
 $(\#) \text{set } \# (\# \# (\# \# \#)) \text{set}) "}$
 $\text{INSERT_EDGE} \text{ " : * \# (\# \# \#) \rightarrow ((\#) \text{set } \# (\# \# (\# \# \#)) \text{set} \rightarrow$
 $(\#) \text{set } \# (\# \# (\# \# \#)) \text{set}) "}$
 $\text{G_INTER} \text{ " : (\#) \text{set } \# (\# \# (\# \# \#)) \text{set} \rightarrow$
 $((\#) \text{set } \# (\# \# (\# \# \#)) \text{set} \rightarrow (\#) \text{set } \# (\# \# (\# \# \#)) \text{set}) "}$
 $\text{G_UNION} \text{ " : (\#) \text{set } \# (\# \# (\# \# \#)) \text{set} \rightarrow$
 $((\#) \text{set } \# (\# \# (\# \# \#)) \text{set} \rightarrow (\#) \text{set } \# (\# \# (\# \# \#)) \text{set}) "}$

```

e_src      ":(* * (* * **))set -> a"
e_des      ":(* * (* * **))set -> a"
eib        ":(* * (* * **))set -> **"
GRAPH      ":(*)set * (* * (* * **))set -> bool"
NULL_GRAPH ":(*)set * (* * (* * **))set"
VS         ":(*)set * (* * (* * **))set -> (* * set)"
ES         ":(*)set * (* * (* * **))set ->
(* * (* * **))set"
IGRAPH     ":(*)set * (* * (* * **))set -> bool"
LOOP       ":(* * (* * **))set -> bool"
HAS_LOOP   ":(*)set * (* * (* * **))set -> bool"
MULTI_EDGE ":(*)set * (* * (* * **))set -> bool"
SIMPLE_GRAPH ":(*)set * (* * (* * **))set -> bool"
FINITE_GRAPH ":(*)set * (* * (* * **))set -> bool"
VER_ADJA   ":(*)set * (* * (* * **))set ->
(* -> (* -> bool))"
E_ADJA     ":(*)set * (* * (* * **))set ->
(* * (* * **))set -> (* * (* * **))set -> bool)"
INCIDENT_FROM ":(*)set * (* * (* * **))set ->
(* -> (* * (* * **))set)"
OUT_DEGREE  ":(*)set * (* * (* * **))set -> (* -> num)"
INCIDENT_TO ":(*)set * (* * (* * **))set ->
(* -> (* * (* * **))set)"
IN_DEGREE  ":(*)set * (* * (* * **))set -> (* -> num)"
INCIDENT_WITH ":(*)set * (* * (* * **))set ->
(* -> (* * (* * **))set)"
DEGREE     ":(*)set * (* * (* * **))set -> (* -> num)"
IS_SUC_VER ":(*)set * (* * (* * **))set ->
(* -> (* -> bool))"
IS_PRE_VER ":(*)set * (* * (* * **))set ->
(* -> (* -> bool))"
SUC_VERS   ":(*)set * (* * (* * **))set ->
(* -> (* * set))"
PRE_VERS   ":(*)set * (* * (* * **))set ->
(* -> (* * set))"

```

```

EDGES.BETWEEN ":(*)set S (* S (* S **))set ->
  (* -> (* -> (* S (* S **))set))"
SUBGRAPH ":(*)set S (* S (* S **))set ->
  ((*set S (* S (* S **))set -> bool)"
PSUBGRAPH ":(*)set S (* S (* S **))set ->
  ((*set S (* S (* S **))set -> bool)"
MK_SUBGRAPH ":(*)set S (* S (* S **))set -> ((* -> bool) ->
  ((* S (* S **)) -> bool) -> (*set S (* S (* S **))set))"
GRAPH.ISO ":(*)set S (* S (* S **))set ->
  ((*set S (* S (* S **))set ->
    ((* -> *) S (* S (* S **)) -> * S (* S **)) -> bool))"

```

Infixes

```

IS.EDGE "::* S (* S **) -> ((*set S (* S (* S **))set -> bool)"
IS.VERTEX "::* -> ((*set S (* S (* S **))set -> bool)"
DELETE.EDGE ":(*)set S (* S (* S **))set ->
  (* S (* S **) -> (*set S (* S (* S **))set)"
DELETE.VERTEX ":(*)set S (* S (* S **))set ->
  (* -> (*set S (* S (* S **))set)"
INSERT.VERTEX "::* -> ((*set S (* S (* S **))set ->
  (*set S (* S (* S **))set)"
INSERT.EDGE "::* S (* S **) ->
  ((*set S (* S (* S **))set -> (*set S (* S (* S **))set)"
G.JINTER ":(*)set S (* S (* S **))set ->
  ((*set S (* S (* S **))set -> (*set S (* S (* S **))set)"
G.UNION ":(*)set S (* S (* S **))set ->
  ((*set S (* S (* S **))set -> (*set S (* S (* S **))set)"

```

Definitions

```

e_src_DEF  $\vdash \forall e. e\_src = FST\ e$ 
e_dst_DEF  $\vdash \forall e. e\_dst = FST\ (SND\ e)$ 
e_lb_DEF  $\vdash \forall e. e\_lb = SND\ (SND\ e)$ 
GRAPH_DEF  $\vdash \forall V\ E. GRAPH(V, E) = \{\forall e. e \in E \supset e\_src \in V \wedge e\_dst \in V\}$ 
NULL_GRAPH  $\vdash NULL\_GRAPH = \{\}, \{\}$ 
VS_DEF  $\vdash \forall G. VS\ G = FST\ G$ 

```

ES_DEF $\vdash \forall G. ES\ G = SND\ G$
IS_EDGE_DEF $\vdash \forall e\ G. e\ IS_EDGE\ G = e\ IN\ ES\ G$
IS_VERTEX_DEF $\vdash \forall v\ G. v\ IS_VERTEX\ G = v\ IN\ VS\ G$
IGRAPH_DEF $\vdash \forall V\ E. IGRAPH\ (V, E) = IMAGE\ a_src\ E\ SUBSET\ V\ \wedge$
 $IMAGE\ a_des\ E\ SUBSET\ V$
LOOP_DEF $\vdash \forall e. LOOP\ e = (a_src\ e = a_des\ e)$
HAS_LOOP_DEF $\vdash \forall G. HAS_LOOP\ G = (\exists e. e\ IN\ ES\ G\ \wedge\ LOOP\ e)$
MULTI_EDGE_DEF $\vdash \forall G. MULTI_EDGE\ G = (\exists e_1\ e_2. e_1\ IN\ ES\ G\ \wedge\ e_2\ IN\ ES\ G\ \wedge$
 $\neg(e_1 = e_2) \wedge (a_src\ e_1 = a_src\ e_2) \wedge (a_des\ e_1 = a_des\ e_2))$
SIMPLE_GRAPH_DEF $\vdash \forall G. SIMPLE_GRAPH\ G = GRAPH\ G\ \wedge\ \neg HAS_LOOP\ G\ \wedge$
 $\neg MULTI_EDGE\ G$
FINITE_GRAPH_DEF
 $\vdash \forall G. FINITE_GRAPH\ G = GRAPH\ G\ \wedge\ FINITE\ (VS\ G) \wedge FINITE\ (ES\ G)$
VER_ADJA_DEF $\vdash \forall G\ v_1\ v_2. VER_ADJA\ G\ v_1\ v_2 = GRAPH\ G\ \wedge\ v_1\ IS_VERTEX\ G\ \wedge$
 $v_2\ IS_VERTEX\ G\ \wedge\ (\exists e. e\ IS_EDGE\ G\ ((a_src\ e = v_1) \wedge (a_des\ e = v_2)) \vee$
 $(a_src\ e = v_2) \wedge (a_des\ e = v_1)))$
E_ADJA_DEF $\vdash \forall G\ e_1\ e_2. E_ADJA\ G\ e_1\ e_2 = GRAPH\ G\ \wedge\ e_1\ IS_EDGE\ G\ \wedge$
 $e_2\ IS_EDGE\ G\ \wedge\ ((a_des\ e_1 = a_src\ e_2) \vee (a_des\ e_2 = a_src\ e_1))$
INCIDENT_FROM_DEF
 $\vdash \forall G\ v. INCIDENT_FROM\ G\ v = \{e \mid e\ IS_EDGE\ G\ \wedge\ (a_src\ e = v)\}$
OUT_DEGREE_DEF $\vdash \forall G\ v. OUT_DEGREE\ G\ v = CARD\ (INCIDENT_FROM\ G\ v)$
INCIDENT_TO_DEF $\vdash \forall G\ v. INCIDENT_TO\ G\ v = \{e \mid e\ IS_EDGE\ G\ \wedge\ (a_des\ e = v)\}$
IN_DEGREE_DEF $\vdash \forall G\ v. IN_DEGREE\ G\ v = CARD\ (INCIDENT_TO\ G\ v)$
INCIDENT_WITH_DEF $\vdash \forall G\ v. INCIDENT_WITH\ G\ v =$
 $\{e \mid e\ IS_EDGE\ G\ \wedge\ ((a_src\ e = v) \vee (a_des\ e = v))\}$
DEGREE_DEF $\vdash \forall G\ v. DEGREE\ G\ v = IN_DEGREE\ G\ v + OUT_DEGREE\ G\ v$
IS_SUC_VER_DEF $\vdash \forall G\ v_1\ v_2. IS_SUC_VER\ G\ v_1\ v_2 = (\exists e. e\ IS_EDGE\ G\ \wedge$
 $(a_src\ e = v_1) \wedge (a_des\ e = v_2))$
IS_PRE_VER_DEF $\vdash \forall G\ v_1\ v_2. IS_PRE_VER\ G\ v_1\ v_2 = (\exists e. e\ IS_EDGE\ G\ \wedge$
 $(a_des\ e = v_1) \wedge (a_src\ e = v_2))$
SUC_VERS_DEF
 $\vdash \forall G\ v. SUC_VERS\ G\ v = \{v' \mid v'\ IS_VERTEX\ G\ \wedge\ IS_SUC_VER\ G\ v\ v'\}$
PRE_VERS_DEF
 $\vdash \forall G\ v. PRE_VERS\ G\ v = \{v' \mid v'\ IS_VERTEX\ G\ \wedge\ IS_PRE_VER\ G\ v\ v'\}$
EDGES_BETWEEN_DEF $\vdash \forall G\ v_1\ v_2. EDGES_BETWEEN\ G\ v_1\ v_2 =$
 $\{e \mid e\ IS_EDGE\ G\ \wedge\ (a_src\ e = v_1) \wedge (a_des\ e = v_2)\}$

$\text{DELETE_EDGE_DEF} \vdash \forall G e. G \text{ DELETE_EDGE } e = \text{VS } G, \text{ES } G \text{ DELETE } e$
 $\text{DELETE_VERTEX_DEF} \vdash \forall G v. G \text{ DELETE_VERTEX } v =$
 $\quad \text{VS } G \text{ DELETE } v, \text{ES } G \text{ DIFF_INTER_WITH } G v$
 $\text{INSERT_VERTEX_DEF} \vdash \forall v G. v \text{ INSERT_VERTEX } G = v \text{ INSERT VS } G, \text{ES } G$
 $\text{INSERT_EDGE_DEF} \vdash \forall e G. e \text{ INSERT_EDGE } G = \text{VS } G, ((e.\text{src } e \text{ IS_VERTEX } G \wedge$
 $\quad e.\text{des } e \text{ IS_VERTEX } G) \Rightarrow e \text{ INSERT ES } G \mid \text{ES } G)$
 $\text{G_INTER_DEF} \vdash \forall G_1 G_2. G_1 \text{ G_INTER } G_2 = \text{VS } G_1 \text{ INTER VS } G_2, \text{ES } G_1 \text{ INTER ES } G_2$
 $\text{G_UNION_DEF} \vdash \forall G_1 G_2. G_1 \text{ G_UNION } G_2 = \text{VS } G_1 \text{ UNION VS } G_2, \text{ES } G_1 \text{ UNION ES } G_2$
 $\text{SUBGRAPH_DEF} \vdash \forall H G. \text{SUBGRAPH } H G = \text{GRAPH } H \wedge \text{GRAPH } G \wedge$
 $\quad \text{VS } H \text{ SUBSET VS } G \wedge \text{ES } H \text{ SUBSET ES } G$
 $\text{PSUBGRAPH_DEF} \vdash \forall H G. \text{PSUBGRAPH } H G = \text{SUBGRAPH } H G \wedge$
 $\quad (\text{VS } H \text{ PSUBSET VS } G \vee \text{ES } H \text{ PSUBSET ES } G)$
 $\text{MK_SUBGRAPH_DEF} \vdash \forall G f v fe. \text{MK_SUBGRAPH } G f v fe =$
 $\quad \{v \mid v \text{ IS_VERTEX } G \wedge f v v\},$
 $\quad \{e \mid e \text{ IS_EDGE } G \wedge fe e \wedge f v (e.\text{src } e) \wedge f v (e.\text{des } e)\}$
 $\text{GRAPH_ISO_DEF} \vdash \forall G H f g. \text{GRAPH_ISO } G H (f, g) = \text{GRAPH } G \wedge \text{GRAPH } H \wedge$
 $\quad \S \langle \cdot \rangle \rightarrow (\text{VS } G) (\text{VS } H) f \wedge \S \langle \cdot \rangle \rightarrow (\text{ES } G) (\text{ES } H) g$

Theorems

$\text{e_src} \vdash \forall p_1 p_2 s. \text{e_src } (p_1, p_2, s) = p_1$
 $\text{e_des} \vdash \forall p_1 p_2 s. \text{e_des } (p_1, p_2, s) = p_2$
 $\text{elb} \vdash \forall p_1 p_2 s. \text{elb } (p_1, p_2, s) = s$
 $\text{VERTICES} \vdash \forall V E. \text{VS } (V, E) = V$
 $\text{EDGES} \vdash \forall V E. \text{ES } (V, E) = E$
 $\text{GRAPH_EQUIV} \vdash \forall V E. \text{GRAPH } (V, E) = \text{IGRAPH } (V, E)$
 $\text{GRAPH_EXISTS} \vdash \exists G. \text{GRAPH } G$
 $\text{GRAPH_PAIR} \vdash \forall G. \text{GRAPH } G \supset (G = \text{VS } G, \text{ES } G)$
 $\text{GRAPH_DECOMP} \vdash \forall G. \text{GRAPH } G = \text{GRAPH } (\text{VS } G, \text{ES } G)$
 $\text{GRAPH_EQ} \vdash \forall G H. \text{GRAPH } G \wedge \text{GRAPH } H \supset ((G = H) = (\text{VS } G = \text{VS } H) \wedge$
 $\quad (\text{ES } G = \text{ES } H))$
 $\text{NOT_VERTEX_NOT_EDGE} \vdash \forall G v_1 v_2 x. \text{GRAPH } G \supset (\neg v_1 \text{ IS_VERTEX } G \vee$
 $\quad \neg v_2 \text{ IS_VERTEX } G \supset \neg (v_1, v_2, x) \text{ IS_EDGE } G)$
 $\text{GRAPH_NOT_VERTEX_NOT_EDGE} \vdash \forall G v. \text{GRAPH } G \wedge \neg v \text{ IS_VERTEX } G \supset (\forall u x. \neg (v, u, x) \text{ IS_EDGE } G)$

GRAPH_NOT_VERTEX_NOT_EDGE2

$\vdash \forall G v. \text{GRAPH } G \wedge \neg v \text{ IS_VERTEX } G \supset (\forall u x. \neg(u, v, x) \text{ IS_EDGE } G)$

EDGE_EQ

$\vdash \forall e_1 e_2. (e_1 = e_2) = (\text{a_src } e_1 = \text{a_src } e_2) \wedge (\text{a_des } e_1 = \text{a_des } e_2) \wedge$
 $(\text{elb } e_1 = \text{elb } e_2)$

GRAPH_DIRECTED

$\vdash \forall G. \text{SIMPLE_GRAPH } G \supset (\forall e_1 e_2. e_1 \text{ IN_ES } G \wedge e_2 \text{ IN_ES } G \wedge$
 $(\text{a_src } e_1 = \text{a_des } e_2) \wedge (\text{a_des } e_1 = \text{a_src } e_2) \supset \neg(e_1 = e_2))$

VER_INCIDENT_NOT_EMPTY

$\vdash \forall G v. \text{GRAPH } G \wedge \neg(\text{INCIDENT_WITH } G v = \{\}) \supset v \text{ IS_VERTEX } G$

NOT_VER_INCIDENT_EMPTY

$\vdash \forall G v. \text{GRAPH } G \supset (\neg v \text{ IS_VERTEX } G \supset (\text{INCIDENT_WITH } G v = \{\}))$

GRAPH_EDGE_VERTEX

$\vdash \forall G e. \text{GRAPH } G \wedge e \text{ IS_EDGE } G \supset \text{a_src } e \text{ IS_VERTEX } G \wedge$
 $\text{a_des } e \text{ IS_VERTEX } G$

NOT_IN_SAME_SET

$\vdash \forall x y a. y \text{ IN } a \wedge \neg x \text{ IN } a \supset \neg(x = y)$

NOT_IN_SAME_GRAPH

$\vdash \forall G v e. \text{GRAPH } G \wedge \neg v \text{ IS_VERTEX } G \wedge$
 $e \text{ IS_EDGE } G \supset \neg(\text{a_src } e = v) \wedge \neg(\text{a_des } e = v)$

VERTEX_EDGE

$\vdash \forall G v e. \text{GRAPH } G \wedge v \text{ IS_VERTEX } G \wedge$
 $e \text{ IN_INCIDENT_WITH } G v \supset (\text{a_src } e = v) \vee (\text{a_des } e = v)$

E_DELETE_ABSORB

$\vdash \forall G e. \text{GRAPH } G \wedge \neg e \text{ IS_EDGE } G \supset (G \text{ DELETE_EDGE } e = G)$

V_DELETE_ABSORB

$\vdash \forall G v. \text{GRAPH } G \wedge \neg v \text{ IS_VERTEX } G \supset (G \text{ DELETE_VERTEX } v = G)$

GRAPH_DELETE_EDGE

$\vdash \forall G e. \text{GRAPH } G \supset \text{GRAPH } (G \text{ DELETE_EDGE } e)$

GRAPH_DELETE_VERTEX

$\vdash \forall G v. \text{GRAPH } G \supset \text{GRAPH } (G \text{ DELETE_VERTEX } v)$

DELETE_VERTEX_COMM

$\vdash \forall G v_1 v_2. (G \text{ DELETE_VERTEX } v_1) \text{ DELETE_VERTEX } v_2 =$
 $(G \text{ DELETE_VERTEX } v_2) \text{ DELETE_VERTEX } v_1$

DELETE_EDGE_COMM

$\vdash \forall G e_1 e_2. (G \text{ DELETE_EDGE } e_1) \text{ DELETE_EDGE } e_2 =$
 $(G \text{ DELETE_EDGE } e_2) \text{ DELETE_EDGE } e_1$

GRAPH_INSERT_VERTEX

$\vdash \forall G v. \text{GRAPH } G \supset \text{GRAPH } (v \text{ INSERT_VERTEX } G)$

GRAPH_INSERT_EDGE

$\vdash \forall G e. \text{GRAPH } G \supset \text{GRAPH } (e \text{ INSERT_EDGE } G)$

INSERT_VERTEX_COMM

$\vdash \forall G v_1 v_2. v_1 \text{ INSERT_VERTEX } (v_2 \text{ INSERT_VERTEX } G) =$
 $v_2 \text{ INSERT_VERTEX } (v_1 \text{ INSERT_VERTEX } G)$

INSERT_EDGE_COMM

$\vdash \forall G e_1 e_2. e_1 \text{ INSERT_EDGE } (e_2 \text{ INSERT_EDGE } G) =$
 $e_2 \text{ INSERT_EDGE } (e_1 \text{ INSERT_EDGE } G)$

IN_INSERT_VERTEX

$\vdash \forall v G. v \text{ IS_EDGE } G \supset e \text{ IS_EDGE } (v \text{ INSERT_VERTEX } G)$

IN_INSERT_EDGE

$\vdash \forall e' G. e \text{ IS_EDGE } G \supset e' \text{ IS_EDGE } (e' \text{ INSERT_EDGE } G)$

INCIDENT_WITH_INSERT_VERTEX

$\vdash \forall G v. \text{GRAPH } G \supset$
 $(\neg v \text{ IS_VERTEX } G \supset (\text{INCIDENT_WITH } (v \text{ INSERT_VERTEX } G) v = \{\}))$

$\text{DELETE_INSERT_EDGE} \vdash \forall G e. \text{GRAPH } G \wedge$
 $e \text{ IS_EDGE } G \supset (e \text{ INSERT_EDGE } (G \text{ DELETE_EDGE } e) = G)$
 $\text{INSERT_DELETE_VERTEX} \vdash \forall G v. \text{GRAPH } G \wedge$
 $\neg v \text{ IS_VERTEX } G \supset ((v \text{ INSERT_VERTEX } G) \text{ DELETE_VERTEX } v = G)$
 $\text{VERTICES_INSERT_EDGE} \vdash \forall G e. \text{VS}(e \text{ INSERT_EDGE } G) = \text{VS } G$
 $\text{EDGES_INSERT_VERTEX} \vdash \forall G v. \text{ES}(v \text{ INSERT_VERTEX } G) = \text{ES } G$
 $\text{VERTEX_INSERT_VERTEX} \vdash \forall G x y. z \text{ IS_VERTEX } (y \text{ INSERT_VERTEX } G) = (z = y) \vee$
 $x \text{ IS_VERTEX } G$
 $\text{EDGE_INSERT_EDGE} \vdash \forall G e. a. \text{src } e \text{ IS_VERTEX } G \wedge$
 $e. \text{dest } e \text{ IS_VERTEX } G \supset e \text{ IS_EDGE } (e \text{ INSERT_EDGE } G)$
 $\text{EDGE_IN_INSERT} \vdash \forall G e. a. \text{src } e \text{ IS_VERTEX } G \wedge$
 $e. \text{dest } e \text{ IS_VERTEX } G \supset e \text{ IS_EDGE } (e \text{ INSERT_EDGE } G)$
 $\text{EDGE_IN_INSERT2} \vdash \forall G v_1 v_2 z. v_1 \text{ IS_VERTEX } G \wedge$
 $v_2 \text{ IS_VERTEX } G \supset (v_1, v_2, z) \text{ IS_EDGE } ((v_1, v_2, z) \text{ INSERT_EDGE } G)$
 $\text{VERTEX_IN_INS_VERTEX} \vdash \forall G v u. v \text{ IS_VERTEX } (u \text{ INSERT_VERTEX } G) = (v = u) \vee$
 $v \text{ IS_VERTEX } G$
 $\text{V_INSERT_ABSORP} \vdash \forall G v. \text{GRAPH } G \wedge v \text{ IS_VERTEX } G \supset (v \text{ INSERT_VERTEX } G = G)$
 $\text{E_INSERT_ABSORP} \vdash \forall G e. \text{GRAPH } G \wedge e \text{ IS_EDGE } G \supset (e \text{ INSERT_EDGE } G = G)$
 $\text{FINITE_GRAPH_INSERT_EDGE} \vdash \forall G e. \text{FINITE_GRAPH } G \supset \text{FINITE_GRAPH } (e \text{ INSERT_EDGE } G)$
 $\text{GRAPH_INTER} \vdash \forall G_1 G_2. \text{GRAPH } G_1 \wedge \text{GRAPH } G_2 \supset \text{GRAPH } (G_1 \text{ G_INTER } G_2)$
 $\text{G_INTER_IDENT} \vdash \forall G. G \text{ G_INTER } G = G$
 $\text{G_INTER_SYM} \vdash \forall G_1 G_2. G_1 \text{ G_INTER } G_2 = G_2 \text{ G_INTER } G_1$
 $\text{G_INTER_ASSOC} \vdash \forall G_1 G_2 G_3.$
 $(G_1 \text{ G_INTER } G_2) \text{ G_INTER } G_3 = G_1 \text{ G_INTER } (G_2 \text{ G_INTER } G_3)$
 $\text{VERTEX_IN_INTER} \vdash \forall G_1 G_2 v. v \text{ IS_VERTEX } (G_1 \text{ G_INTER } G_2) = v \text{ IS_VERTEX } G_1 \wedge$
 $v \text{ IS_VERTEX } G_2$
 $\text{EDGE_IN_INTER} \vdash \forall G_1 G_2 e. e \text{ IS_EDGE } (G_1 \text{ G_INTER } G_2) = e \text{ IS_EDGE } G_1 \wedge e \text{ IS_EDGE } G_2$
 $\text{GRAPH_UNION} \vdash \forall G_1 G_2. \text{GRAPH } G_1 \wedge \text{GRAPH } G_2 \supset \text{GRAPH } (G_1 \text{ G_UNION } G_2)$
 $\text{G_UNION_IDENT} \vdash \forall G. G \text{ G_UNION } G = G$
 $\text{G_UNION_SYM} \vdash \forall G_1 G_2. G_1 \text{ G_UNION } G_2 = G_2 \text{ G_UNION } G_1$
 $\text{G_UNION_ASSOC} \vdash \forall G_1 G_2 G_3.$
 $(G_1 \text{ G_UNION } G_2) \text{ G_UNION } G_3 = G_1 \text{ G_UNION } (G_2 \text{ G_UNION } G_3)$

$\text{VERTICES_IN_UNION} \vdash \forall G_1 G_2 v_1 v_2. \text{GRAPH } G_1 \wedge \text{GRAPH } G_2 \wedge v_1 \text{ IS_VERTEX } G_1 \wedge$
 $v_2 \text{ IS_VERTEX } G_2 \supset v_1 \text{ IS_VERTEX } (G_1 \text{ G_UNION } G_2) \wedge$
 $v_2 \text{ IS_VERTEX } (G_1 \text{ G_UNION } G_2)$

$\text{VERTEX_IN_UNION} \vdash \forall G_1 G_2 v. v \text{ IS_VERTEX } (G_1 \text{ G_UNION } G_2) = v \text{ IS_VERTEX } G_1 \vee$
 $v \text{ IS_VERTEX } G_2$

EDGE_IN_UNION
 $\vdash \forall G_1 G_2 e. e \text{ IS_EDGE } (G_1 \text{ G_UNION } G_2) = e \text{ IS_EDGE } G_1 \vee e \text{ IS_EDGE } G_2$

$\text{VERTEX_INSERT_EDGE}$
 $\vdash \forall G v e. v \text{ IS_VERTEX } (e \text{ INSERT_EDGE } G) = v \text{ IS_VERTEX } G$

$\text{GRAPH_INSERT_EDGES} \vdash \forall G v_1 v_2. \text{GRAPH } G \wedge v_1 \text{ IS_VERTEX } G \wedge$
 $v_2 \text{ IS_VERTEX } G \supset (\forall x_1. \text{GRAPH } ((v_1, v_2, x_1) \text{ INSERT_EDGE } G)) \wedge$
 $(\forall x_2. \text{GRAPH } ((v_2, v_1, x_2) \text{ INSERT_EDGE } G))$

$\text{G_UNION_INSERT_EDGES} \vdash \forall G_1 G_2 v_1 v_2.$
 $\text{GRAPH } G_1 \wedge \text{GRAPH } G_2 \wedge v_1 \text{ IS_VERTEX } G_1 \wedge v_2 \text{ IS_VERTEX } G_2 \supset$
 $(\forall x_1. \text{GRAPH } ((v_1, v_2, x_1) \text{ INSERT_EDGE } (G_1 \text{ G_UNION } G_2))) \wedge$
 $(\forall x_2. \text{GRAPH } ((v_2, v_1, x_2) \text{ INSERT_EDGE } (G_1 \text{ G_UNION } G_2)))$

$\text{G_INS_INS_E} \vdash \forall G e_1 e_2.$
 $\text{GRAPH } (e_1 \text{ INSERT_EDGE } G) \wedge \text{GRAPH } (e_2 \text{ INSERT_EDGE } G) \supset$
 $\text{GRAPH } (e_1 \text{ INSERT_EDGE } (e_2 \text{ INSERT_EDGE } G)) \wedge$
 $\text{GRAPH } (e_2 \text{ INSERT_EDGE } (e_1 \text{ INSERT_EDGE } G))$

$\text{G_UNION_INS_EDGES} \vdash \forall G_1 G_2 v_1 v_2. \text{GRAPH } G_1 \wedge \text{GRAPH } G_2 \wedge$
 $v_1 \text{ IS_VERTEX } G_1 \wedge v_2 \text{ IS_VERTEX } G_2 \supset$
 $(\forall x_2 x_1. \text{GRAPH } ((v_1, v_2, x_1) \text{ INSERT_EDGE } ((v_2, v_1, x_2) \text{ INSERT_EDGE } (G_1 \text{ G_UNION } G_2)))) \wedge$
 $\text{GRAPH } ((v_2, v_1, x_2) \text{ INSERT_EDGE } ((v_1, v_2, x_1) \text{ INSERT_EDGE } (G_1 \text{ G_UNION } G_2))))$

$\text{SUBGRAPH_REFL} \vdash \forall G. \text{GRAPH } G \supset \text{SUBGRAPH } G G$

$\text{SUBGRAPH_TRANS} \vdash \forall G_1 G_2 G_3. \text{SUBGRAPH } G_1 G_2 \wedge$
 $\text{SUBGRAPH } G_2 G_3 \supset \text{SUBGRAPH } G_1 G_3$

$\text{SUBGRAPH_ANTISYM} \vdash \forall G_1 G_2. \text{SUBGRAPH } G_1 G_2 \wedge \text{SUBGRAPH } G_2 G_1 \supset (G_1 = G_2)$

$\text{SUBGRAPH_GRAPH} \vdash \forall G H. \text{SUBGRAPH } H G \supset \text{GRAPH } G \wedge \text{GRAPH } H$

$\text{PSUBGRAPH_SUBGRAPH} \vdash \forall G H. \text{PSUBGRAPH } H G \supset \text{SUBGRAPH } H G$

$\text{PSUBGRAPH_IRREFL} \vdash \forall G. \text{GRAPH } G \supset \neg \text{PSUBGRAPH } G G$

$\text{PSUBGRAPH_TRANS} \vdash \forall G_1 G_2 G_3. \text{PSUBGRAPH } G_1 G_2 \wedge$
 $\text{PSUBGRAPH } G_2 G_3 \supset \text{PSUBGRAPH } G_1 G_3$

$\text{PSUBGRAPH_DELETE_EDGE}$
 $\vdash \forall G e. \text{GRAPH } G \wedge e \text{ IS_EDGE } G \supset \text{PSUBGRAPH } (G \text{ DELETE_EDGE } e) G$

$\text{SUBGRAPH_DELETE_EDGE} \vdash \forall G e. \text{GRAPH } G \supset \text{SUBGRAPH } (G \text{ DELETE_EDGE } e) G$

SUBGRAPH_DELETE_VERTEX
 $\vdash \forall G v. \text{GRAPH } G \supset \text{SUBGRAPH } (G \text{ DELETE_VERTEX } v) G$
 PSUBGRAPH_DELETE_VERTEX $\vdash \forall G v. \text{GRAPH } G \wedge$
 $v \text{ IS_VERTEX } G \supset \text{PSUBGRAPH } (G \text{ DELETE_VERTEX } v) G$
 MK.SUBGRAPH_GRAPH $\vdash \forall G f v f e. \text{GRAPH } G \supset \text{GRAPH } (\text{MK.SUBGRAPH } G f v f e)$
 MK.SUBGRAPH_SUBGRAPH
 $\vdash \forall G f v f e. \text{GRAPH } G \supset \text{SUBGRAPH } (\text{MK.SUBGRAPH } G f v f e) G$
 GRAPH_ISO_AUTO $\vdash \forall G. \text{GRAPH } G \supset \text{GRAPH.ISO } G G (I, I)$
 GRAPH_ISO_TRANS $\vdash \forall G_1 G_2 f_1 g_1 f_2 g_2. \text{GRAPH.ISO } G_1 G_2 (f_1, g_1) \wedge$
 $\text{GRAPH.ISO } G_2 G_3 (f_2, g_2) \supset \text{GRAPH.ISO } G_1 G_3 ((f_1 \circ f_2), (g_2 \circ g_1))$
 GRAPH_ISO_SYM
 $\vdash \forall G H f g. \text{GRAPH.ISO } G H (f, g) \supset (\exists f' g'. \text{GRAPH.ISO } H G (f', g'))$
 GRAPH_ISO_SYM_INV $\vdash \forall G H f g. \text{GRAPH.ISO } G H (f, g) \supset$
 $\text{GRAPH.ISO } H G (\text{FUN_INV } (VS G) (VS H) f, \text{FUN_INV } (ES G) (ES H) g)$

End of theory graph

A.3 The theory elist

Parents

HOL sets graph

Constants

ELEM ":(*)list -> (* -> bool)"
 UNIQUE_EL ":(*)list -> bool"
 EL_SET ":(*)list -> (*)set"
 DISJ_LIST ":(*)list -> ((*)list -> bool)"
 V_L ":(* @ (* @ **))list -> (*)list"
 VER_LIST ":(* @ (* @ **))list -> (*)list"

Definitions

ELEM_DEF $\vdash (\forall x. \text{ELEM } [] \ x = F) \wedge (\forall h \ i \ x. \text{ELEM } (\text{CONS } h \ i) \ x = (x = h) \vee \text{ELEM } i \ x)$

UNIQUE_EL_DEF $\vdash (\text{UNIQUE_EL } [] = T) \wedge$
 $(\forall h \ i \ t. \text{UNIQUE_EL } (\text{CONS } h \ i \ t) = \text{EVERY } (\lambda x. \neg(x = h)) \ t) \wedge$
 $\text{UNIQUE_EL } i \ t)$

EL_SET_DEF $\vdash (\text{EL_SET } [] = \{\}) \wedge$
 $(\forall h \ i \ t. \text{EL_SET } (\text{CONS } h \ i \ t) = h \ \text{INSERT } \text{EL_SET } i \ t)$

DISJ_LIST_DEF $\vdash \forall i_1 \ i_2. \text{DISJ_LIST } i_1 \ i_2 = \text{DISJOINT } (\text{EL_SET } i_1) (\text{EL_SET } i_2)$

V_L_DEF $\vdash (\text{V_L } [] = []) \wedge (\forall h \ i \ t. \text{V_L } (\text{CONS } h \ i \ t) = \text{CONS } (e_den \ h) (\text{V_L } i \ t))$

VER_LIST_DEF $\vdash (\text{VER_LIST } [] = []) \wedge$
 $(\forall h \ i \ t. \text{VER_LIST } (\text{CONS } h \ i \ t) = \text{CONS } (e_src \ h) (\text{V_L } (\text{CONS } h \ i \ t)))$

Theorems

NULL_NIL $\vdash \forall i. \text{NULL } i = (i = [])$

NULL_NOT_ELEM $\vdash \forall i. \text{NULL } i \supset (\forall x. \neg \text{ELEM } i \ x)$

ELEM_CONS $\vdash \forall i \ x \ y. \text{ELEM } i \ x \supset \text{ELEM } (\text{CONS } y \ i) \ x$

ELEM_APPEND $\vdash \forall i_1 \ i_2 \ x. \text{ELEM } (\text{APPEND } i_1 \ i_2) \ x = \text{ELEM } i_1 \ x \vee \text{ELEM } i_2 \ x$

ELEM_EL $\vdash \forall i \ x. \text{ELEM } i \ x \supset (\exists n. x = \text{EL } n \ i)$

IN_ELEM $\vdash \forall s. \text{FINITE } s \supset (\exists i. (\forall x. x \ \text{IN } s = \text{ELEM } i \ x))$

UNIQUE_EL_TL $\vdash \forall i \ h. \text{UNIQUE_EL } (\text{CONS } h \ i) \supset \text{UNIQUE_EL } i$

UNIQUE_EL_SIMP $\vdash \forall x. \text{UNIQUE_EL } [x]$

ELEM_NOT_UNIQUE_EL_CONS $\vdash \forall i \ h. \text{ELEM } i \ h \supset \neg \text{UNIQUE_EL } (\text{CONS } h \ i)$

NOT_ELEM_UNIQUE_EL_CONS

$\vdash \forall i \ h. \text{UNIQUE_EL } i \ h \wedge \neg \text{ELEM } i \ h \supset \text{UNIQUE_EL } (\text{CONS } h \ i)$

EL_SET_APPEND $\vdash \forall i_1 \ i_2. \text{EL_SET } (\text{APPEND } i_1 \ i_2) = \text{EL_SET } i_1 \ \text{UNION } \text{EL_SET } i_2$

ELEM_IN_EL_SET $\vdash \forall i \ x. \text{ELEM } i \ x = x \ \text{IN } \text{EL_SET } i$

DISJ_LIST_EMPTY $\vdash \forall i. \text{DISJ_LIST } [] \ i \wedge \text{DISJ_LIST } i \ []$

DISJ_LIST_CONS $\vdash \forall i_1 \ i_2 \ h. \text{DISJ_LIST } (\text{CONS } h \ i_1) \ i_2 = \text{DISJ_LIST } i_1 \ i_2 \wedge \neg \text{ELEM } i_2 \ h$

DISJ_LIST_APPEND

$\vdash \forall i_1 \ i_2 \ i_3. \text{DISJ_LIST } (\text{APPEND } i_1 \ i_2) \ i_3 = \text{DISJ_LIST } i_1 \ i_3 \wedge \text{DISJ_LIST } i_2 \ i_3$

DISJ_LIST_COMM $\vdash \forall i_1 \ i_2. \text{DISJ_LIST } i_1 \ i_2 = \text{DISJ_LIST } i_2 \ i_1$

V_L_APPEND $\vdash \forall p_1 \ p_2. \text{V_L } (\text{APPEND } p_1 \ p_2) = \text{APPEND } (\text{V_L } p_1) (\text{V_L } p_2)$

NOT_NULL_VER_LIST $\vdash \forall p. \neg \text{NULL } p \supset (\text{VER_LIST } p = \text{CONS } (e_src \ (\text{HD } p)) (\text{V_L } p))$

VER_LIST_CONS
 $\vdash \forall p\ h. VER_LIST\ (CONS\ h\ p) = CONS\ (e_arc\ h)\ (CONS\ (e_den\ h)\ (V\bot\ p))$
 $NOT_NULL_VER_LIST_CONS\ \vdash \forall i\ h. \neg NULL\ i \wedge (e_den\ h = e_arc\ (HD\ i)) \supset$
 $(VER_LIST\ (CONS\ h\ i) = CONS\ (e_arc\ h)\ (VER_LIST\ i))$
 $TL_VER_LIST\ \vdash \forall p. \neg NULL\ p \supset (TL\ (VER_LIST\ p) = V\bot\ p)$
 $VER_LIST_APPEND\ \vdash \forall p_1\ p_2. \neg NULL\ p_1 \wedge \neg NULL\ p_2 \supset (VER_LIST\ (APPEND\ p_1\ p_2) =$
 $APPEND\ (VER_LIST\ p_1)\ (TL\ (VER_LIST\ p_2)))$
 $UNIQUE_EL_CONS\ \vdash \forall i\ h. UNIQUE_EL\ i \wedge \neg h\ IN\ EL_SET\ i \supset UNIQUE_EL\ (CONS\ h\ i)$
 $NOT_UNIQUE_EL_CONS\ \vdash \forall i\ h. h\ IN\ EL_SET\ i \supset (UNIQUE_EL\ (CONS\ h\ i) = F)$
 $UNIQUE_EL_APPEND\ \vdash \forall i_1\ i_2. UNIQUE_EL\ (APPEND\ i_1\ i_2) = UNIQUE_EL\ i_1 \wedge$
 $UNIQUE_EL\ i_2 \wedge DISJ_LIST\ i_1\ i_2$
 $UNIQUE_V\bot_CONS\ \vdash \forall p\ h. UNIQUE_EL\ (V\bot\ p) \wedge$
 $\neg ELEM\ (V\bot\ p)\ (e_den\ h) \supset UNIQUE_EL\ (V\bot\ (CONS\ h\ p))$
 $UNIQUE_VER_LIST_CONS\ \vdash \forall p\ h. \neg NULL\ p \wedge UNIQUE_EL\ (VER_LIST\ p) \wedge$
 $(e_arc\ (HD\ p) = e_den\ h) \wedge \neg LOOP\ h \wedge$
 $\neg ELEM\ (VER_LIST\ p)\ (e_arc\ h) \supset UNIQUE_EL\ (VER_LIST\ (CONS\ h\ p))$
 $UNIQUE_EL_VER_LIST_TL\ \vdash \forall p. \neg NULL\ p \supset (UNIQUE_EL\ (VER_LIST\ p) \supset$
 $UNIQUE_EL\ (TL\ (VER_LIST\ p)))$
 $UNIQUE_VER_LIST_APPEND\ \vdash \forall p_1\ p_2\ G. \neg NULL\ p_1 \wedge \neg NULL\ p_2 \supset$
 $(UNIQUE_EL\ (VER_LIST\ p_1) \wedge UNIQUE_EL\ (VER_LIST\ p_2) \wedge$
 $DISJ_LIST\ (V\bot\ p_1)\ (V\bot\ p_2) \wedge \neg ELEM\ (VER_LIST\ p_2)\ (e_arc\ (HD\ p_1)) \supset$
 $UNIQUE_EL\ (VER_LIST\ (APPEND\ p_1\ p_2)))$

End of theory alist

A.4 The theory path

Parents

HOL sets graph alist

Constants

```

WALK.TAIL ":(e $ (e $ ee))list ->
            ((e)set $ (e $ (e $ ee))set -> bool)"
WALK      ":(e)set $ (e $ (e $ ee))set ->
            ((e $ (e $ ee))list -> bool)"
WALK.ENTRY ":(e $ (e $ ee))list -> e"
WALK.EXIT  ":(e $ (e $ ee))list -> e"
TRAIL      ":(e)set $ (e $ (e $ ee))set ->
            ((e $ (e $ ee))list -> bool)"
PATH       ":(e)set $ (e $ (e $ ee))set ->
            ((e $ (e $ ee))list -> bool)"
PATH.ENTRY ":(e $ (e $ ee))list -> e"
PATH.EXIT  ":(e $ (e $ ee))list -> e"
CONNECTED  ":(e)set $ (e $ (e $ ee))set -> bool"
DISJ.PATH  ":(e)set $ (e $ (e $ ee))set ->
            ((e $ (e $ ee))list -> ((e $ (e $ ee))list -> bool)))"
HAS.PATH   ":(e)set $ (e $ (e $ ee))set -> (e -> (e -> bool))"

```

Definitions

```

WALK.TAIL_DEF  $\vdash \forall G. \text{WALK.TAIL}[G] = T \wedge$ 
                $(\forall G \text{ hd } tl. \text{WALK.TAIL}(\text{CONS } hd \text{ tl}) G = \text{GRAPH } G \wedge \text{hd IS\_EDGE } G \wedge$ 
                $(\text{NULL } tl \vee \text{WALK.TAIL } tl G \wedge (e.\text{des } hd = e.\text{src } (HD \text{ tl}))))$ 
WALK_DEF  $\vdash \forall G w. \text{WALK } G w = \neg \text{NULL } w \wedge \text{WALK.TAIL } w G$ 
WALK.ENTRY_DEF  $\vdash \forall l. \text{WALK.ENTRY } l = e.\text{src } (HD \text{ l})$ 
WALK.EXIT_DEF  $\vdash \forall hd \text{ tl}. \text{WALK.EXIT}(\text{CONS } hd \text{ tl}) =$ 
                $(\text{NULL } tl \Rightarrow e.\text{des } hd \mid \text{WALK.EXIT } tl)$ 
TRAIL_DEF  $\vdash \forall G l. \text{TRAIL } G l = \text{WALK } G l \wedge \text{UNIQUE.ELEMENT } l$ 
PATH_DEF  $\vdash \forall G l. \text{PATH } G l = \text{TRAIL } G l \wedge \text{UNIQUE.ELEMENT } (VER.LIST \text{ l})$ 
PATH.ENTRY_DEF  $\vdash \forall l. \text{PATH.ENTRY } l = e.\text{src } (HD \text{ l})$ 
PATH.EXIT_DEF  $\vdash \forall p. \text{PATH.EXIT } p = \text{WALK.EXIT } p$ 
CONNECTED_DEF  $\vdash \forall G. \text{CONNECTED } G = \text{GRAPH } G \wedge (\forall v_1 v_2. v_1 \text{ IS\_VERTEX } G \wedge$ 
                $v_2 \text{ IS\_VERTEX } G \wedge \neg(v_1 = v_2) \supset (\exists l. \text{PATH } G l \wedge$ 
                $(v_1 = \text{PATH.ENTRY } l) \wedge (v_2 = \text{PATH.EXIT } l)))$ 
DISJ.PATH_DEF  $\vdash \forall G p_1 p_2. \text{DISJ.PATH } G p_1 p_2 = \text{PATH } G p_1 \wedge \text{PATH } G p_2 \wedge$ 
                $\text{DISJ.LIST } p_1 p_2 \wedge \text{DISJ.LIST } (V.L \text{ p}_1) (V.L \text{ p}_2)$ 

```

HAS_PATH_DEF $\vdash \forall G \, v_1 \, v_2. \text{HAS_PATH } G \, v_1 \, v_2 = (\exists p. \text{PATH } G \, p \wedge$
 $(\text{PATH_ENTRY } p = v_1) \wedge (\text{PATH_EXIT } p = v_2))$

Theorems

PATH_TRAIL $\vdash \forall G. \text{PATH } G \, l \supset \text{TRAIL } G \, l$

TRAIL_WALK $\vdash \forall G. \text{TRAIL } G \, l \supset \text{WALK } G \, l$

PATH_WALK $\vdash \forall G. \text{PATH } G \, l \supset \text{WALK } G \, l$

PATH_GRAPH $\vdash \forall G \, l. \text{PATH } G \, l \supset \text{GRAPH } G$

PATH_NOT_NULL $\vdash \forall p \, G. \text{PATH } G \, l \supset \neg \text{NULL } l$

PATH_WALK_ENTRY $\vdash \forall p. \text{PATH_ENTRY } p = \text{WALK_ENTRY } p$

PATH_CONNECTED

$\vdash \forall p \, h \, G. \text{PATH } G (\text{CONS } h \, p) \wedge \neg \text{NULL } p \supset (e.\text{des } h = e.\text{src } (\text{HD } p))$

CONNECTED_GRAPH $\vdash \forall G. \text{CONNECTED } G \supset \text{GRAPH } G$

CONNECTED_SING $\vdash \forall v. \text{CONNECTED } (\{v\}, \{\})$

WALK_ENTRY_CONS $\vdash \forall p \, h \, G. \text{WALK_ENTRY } (\text{CONS } h \, p) = e.\text{src } h$

WALK_ENTRY_APPEND $\vdash \forall p_1 \, p_2 \, G. \text{WALK } G \, p_1 \wedge$

$\text{WALK } G \, p_2 \supset (\text{WALK_ENTRY } (\text{APPEND } p_1 \, p_2) = \text{WALK_ENTRY } p_1)$

WALK_EXIT_APPEND $\vdash \forall p_1 \, p_2 \, G. \text{WALK } G \, p_1 \wedge$

$\text{WALK } G \, p_2 \supset (\text{WALK_EXIT } (\text{APPEND } p_1 \, p_2) = \text{WALK_EXIT } p_2)$

PATH_ENTRY_SIMP $\vdash \forall u \, v \, z. \text{PATH_ENTRY } [u, v, z] = u$

PATH_EXIT_SIMP $\vdash \forall u \, v \, z. \text{PATH_EXIT } [u, v, z] = v$

PATH_ENTRY_CONS $\vdash \forall p \, h. \text{PATH_ENTRY } (\text{CONS } h \, p) = e.\text{src } h$

PATH_EXIT_CONS $\vdash \forall p \, h. \neg \text{NULL } p \supset (\text{PATH_EXIT } (\text{CONS } h \, p) = \text{PATH_EXIT } p)$

PATH_ENTRY_APPEND $\vdash \forall l_1 \, l_2. \text{PATH } G \, l_1 \supset$

$(\text{PATH_ENTRY } (\text{APPEND } l_1 \, l_2) = \text{PATH_ENTRY } l_1)$

PATH_EXIT_APPEND

$\vdash \forall l_1 \, l_2. \neg \text{NULL } l_2 \supset (\text{PATH_EXIT } (\text{APPEND } l_1 \, l_2) = \text{PATH_EXIT } l_2)$

WALK_CONS $\vdash \forall p \, h \, G. \text{WALK } G \, p \wedge h \text{ IS_EDGE } G \wedge$

$(e.\text{des } h = \text{WALK_ENTRY } p) \supset \text{WALK } G (\text{CONS } h \, p)$

WALK_APPEND $\vdash \forall p_1 \, p_2 \, G. \text{WALK } G \, p_1 \wedge \text{WALK } G \, p_2 \wedge$

$(\text{WALK_EXIT } p_1 = \text{WALK_ENTRY } p_2) \supset \text{WALK } G (\text{APPEND } p_1 \, p_2)$

WALK_CAT $\vdash \forall G \, p_1 \, p_2. \text{WALK } G \, p_1 \wedge \text{WALK } G \, p_2 \wedge$

$(\text{WALK_EXIT } p_1 = \text{WALK_ENTRY } p_2) \supset (\exists p_3. \text{WALK } G \, p_3 \wedge$

$(\text{WALK_ENTRY } p_2 = \text{WALK_ENTRY } p_1) \wedge$

$(\text{WALK_EXIT } p_3 = \text{WALK_EXIT } p_2) \wedge (p_3 = \text{APPEND } p_1 \, p_2))$

$\text{PATH_EDGE_NO_LOOP} \vdash \forall p \, h \, G. \text{PATH } G (\text{CONS } h \, p) \supset \neg (e.\text{src } h = e.\text{des } h)$
 $\text{PATH_SIMP} \vdash \forall G \, e. \text{GRAPH } G \wedge e \text{ IS_EDGE } G \wedge \neg \text{LOOP } e \supset \text{PATH } G [e]$
 $\text{PATH_CONS} \vdash \forall p \, h \, G. \text{GRAPH } G \wedge \text{PATH } G \, p \wedge h \text{ IS_EDGE } G \wedge$
 $(\text{PATH_ENTRY } p = e.\text{des } h) \wedge \neg \text{LOOP } h \wedge$
 $\neg \text{ELEM } (\text{VER_LIST } p) (e.\text{src } h) \wedge \neg \text{ELEM } p \, h \supset \text{PATH } G (\text{CONS } h \, p)$
 $\text{PATH_CAT} \vdash \forall G \, p_1 \, p_2. \text{GRAPH } G \wedge \text{DISJ.PATH } G \, p_1 \, p_2 \wedge$
 $(\text{PATH_EXIT } p_1 = \text{PATH_ENTRY } p_2) \wedge$
 $\neg \text{ELEM } (\text{VER_LIST } p_2) (\text{PATH_ENTRY } p_1) \supset (\exists p_3. \text{PATH } G \, p_3 \wedge$
 $(\text{PATH_ENTRY } p_3 = \text{PATH_ENTRY } p_1) \wedge$
 $(\text{PATH_EXIT } p_3 = \text{PATH_EXIT } p_2) \wedge (p_3 = \text{APPEND } p_1 \, p_2))$
 $\text{PATH_APPEND} \vdash \forall G \, p_1 \, p_2. \text{GRAPH } G \wedge \text{DISJ.PATH } G \, p_1 \, p_2 \wedge$
 $(\text{PATH_EXIT } p_1 = \text{PATH_ENTRY } p_2) \wedge$
 $\neg \text{ELEM } (\text{VER_LIST } p_2) (\text{PATH_ENTRY } p_1) \supset \text{PATH } G (\text{APPEND } p_1 \, p_2)$
 $\text{PATH_NOT_NIL} \vdash \forall G. \neg \text{PATH } G []$
 $\text{WALK_TAIL_G_UNION} \vdash \forall I \, G_1 \, G_2. \text{GRAPH } G_1 \wedge \text{GRAPH } G_2 \wedge$
 $\text{WALK.TAIL } I \, G_1 \supset \text{WALK.TAIL } I (G_1 \, G_2.\text{UNION } G_2)$
 $\text{PATH_G_UNION} \vdash \forall I \, G_1 \, G_2. \text{GRAPH } G_1 \wedge \text{GRAPH } G_2 \wedge$
 $\text{PATH } G_1 \, I \supset \text{PATH } (G_1 \, G_2.\text{UNION } G_2) \, I$
 $\text{WALK_TAIL_INS_VERTEX}$
 $\vdash \forall I \, v \, G. \text{WALK.TAIL } I \, G \supset \text{WALK.TAIL } I (v \text{ INSERT_VERTEX } G)$
 $\text{WALK_TAIL_INS_EDGE} \vdash \forall I \, e \, G. \text{WALK.TAIL } I \, G \supset \text{WALK.TAIL } I (e \text{ INSERT_EDGE } G)$
 $\text{PATH_INS_VERTEX} \vdash \forall I \, v \, G. \text{PATH } G \, I \supset \text{PATH } (v \text{ INSERT_VERTEX } G) \, I$
 $\text{PATH_INS_EDGE} \vdash \forall I \, e \, G. \text{PATH } G \, I \supset \text{PATH } (e \text{ INSERT_EDGE } G) \, I$
 $\text{PATH_INS_EDGE2} \vdash \forall G \, v_1 \, v_2. \text{GRAPH } G \wedge v_1 \text{ IS_VERTEX } G \wedge v_2 \text{ IS_VERTEX } G \wedge$
 $\neg (v_1 = v_2) \supset (\forall x. \text{PATH } ((v_1, v_2, x) \text{ INSERT_EDGE } G) [v_1, v_2, x])$
 $\text{PATH_IS_EDGE} \vdash \forall G \, h \, I. \text{PATH } G (\text{CONS } h \, I) \supset h \text{ IS_EDGE } G$
 $\text{PATH_ELEM_IS_EDGE} \vdash \forall G \, I. \text{PATH } G \, I \supset (\forall x. \text{ELEM } I \, x \supset x \text{ IS_EDGE } G)$
 $\text{PATH_IS_VERTEX} \vdash \forall G \, h \, I. \text{PATH } G (\text{CONS } h \, I) \supset e.\text{src } h \text{ IS_VERTEX } G \wedge$
 $e.\text{des } h \text{ IS_VERTEX } G$
 $\text{PATH_ELEM_VER_LIST_IS_VERTEX}$
 $\vdash \forall G \, I. \text{PATH } G \, I \supset (\forall x. \text{ELEM } (\text{VER_LIST } I) \, x \supset x \text{ IS_VERTEX } G)$
 $\text{PATH_INS_INS_CONS} \vdash \forall G \, I \, v_1 \, v_2 \, x.$
 $\text{PATH } G \, I \wedge v_2 \text{ IS_VERTEX } G \wedge \neg v_1 \text{ IS_VERTEX } G \wedge$
 $(v_2 = \text{PATH_ENTRY } I) \wedge \neg (v_1 = v_2) \supset \text{PATH } ((v_1, v_2, x) \text{ INSERT_EDGE } G)$
 $(v_1 \text{ INSERT_VERTEX } G) (\text{CONS } (v_1, v_2, x) \, I)$
 $\text{CONNECTED_INS_EDGE}$
 $\vdash \forall G. \text{CONNECTED } G \supset (\forall e. \text{CONNECTED } (e \text{ INSERT_EDGE } G))$

A.5 The theory SIGNAL

Parents

HOL

Types

```
" :ShAspect" " :Shsig" " :SubAspect" " :Subsig" " :Jsig" " :NAspect"
" :Ntype" " :Nsig" " :Signal"
```

Constants

```
REP_ShAspect " :ShAspect -> (one + (one + one))ltree"
ABS_ShAspect " : (one + (one + one))ltree -> ShAspect"
sh_on " :ShAspect"
sh_off " :ShAspect"
sh_faulty " :ShAspect"
REP_Shsig " :Shsig -> (num -> ShAspect)ltree"
ABS_Shsig " : (num -> ShAspect)ltree -> Shsig"
SHUNTSIG " : (num -> ShAspect) -> Shsig"
SHUNT_FUNC " :Shsig -> (num -> ShAspect)"
SHUNT_ON " :Shsig -> (num -> bool)"
SHUNT_OFF " :Shsig -> (num -> bool)"
SHUNT_FAULT " :Shsig -> (num -> bool)"
REP_SubAspect " :SubAspect -> (one + one)ltree"
ABS_SubAspect " : (one + one)ltree -> SubAspect"
sub_not_show " :SubAspect"
sub_off " :SubAspect"
REP_Subsig " :Subsig -> (num -> SubAspect)ltree"
ABS_Subsig " : (num -> SubAspect)ltree -> Subsig"
SUBSIG " : (num -> SubAspect) -> Subsig"
SUB_FUNC " :Subsig -> (num -> SubAspect)"
SUB_OFF " :Subsig -> (num -> bool)"
```

```

REP_Isig  ":Isig -> (num -> bool)ltree"
ABS_Isig  ": (num -> bool)ltree -> Isig"
JSIG      ": (num -> bool) -> Isig"
JFUNC     ":Isig -> (num -> bool)"
REP_MAspect ":MAspect -> (one + (one + (one +
      (one + (one + (one + (one + one))))))ltree"
ABS_MAspect ": (one + (one + (one + (one + (one +
      (one + (one + one))))))ltree -> MAspect"
green     ":MAspect"
double_yellow ":MAspect"
yellow    ":MAspect"
red       ":MAspect"
green_flash ":MAspect"
double_yellow_flash ":MAspect"
yellow_flash ":MAspect"
faulty_aspect ":MAspect"
REP_Mtype  ":Mtype -> (one + (one + (one + (one + one))))ltree"
ABS_Mtype  ": (one + (one + (one + (one + one))))ltree -> Mtype"
two_aspect ":Mtype"
three_aspect ":Mtype"
four_aspect ":Mtype"
two_repeat ":Mtype"
three_repeat ":Mtype"
REP_Msig   ":Msig -> (Mtype # (num -> MAspect))ltree"
ABS_Msig   ": (Mtype # (num -> MAspect))ltree -> Msig"
MSIG       ":Mtype -> ((num -> MAspect) -> Msig)"
M_TYPE     ":Msig -> Mtype"
M_FUNC     ":Msig -> (num -> MAspect)"
M_ASPECT   ":Msig -> (num -> MAspect)"
MAIN_ON    ":Msig -> (num -> bool)"
MAIN_FAULTY ":Msig -> (num -> bool)"
MAIN_OFF   ":Msig -> (num -> bool)"

```

```

RED      ":Meig -> (num -> bool)"
YELLOW   ":Meig -> (num -> bool)"
REP.Signal ":Signal -> (num @ Meig +
  (num @ (Meig @ Jsig) + (num @ (Meig @ Subsig) +
    (num @ (Meig @ (Subsig @ Jsig)) + num @ Shaig))))ltree"
ABS.Signal ": (num @ Meig + (num @ (Meig @ Jsig) +
  (num @ (Meig @ Subsig) + (num @ (Meig @ (Subsig @ Jsig)) +
    num @ Shaig))))ltree -> Signal"
SIGNALM  ":num -> (Meig -> Signal)"
SIGNALMJ ":num -> (Meig -> (Jsig -> Signal))"
SIGNALMS ":num -> (Meig -> (Subsig -> Signal))"
SIGNALMSJ ":num -> (Meig -> (Subsig -> (Jsig -> Signal)))"
SIGNALS  ":num -> (Shaig -> Signal)"
SIGNAL.ID ":Signal -> num"
SIGNAL.MAIN ":Signal -> Meig"
SIGNAL.JUNC ":Signal -> Jsig"
SIGNAL.SUB ":Signal -> Subsig"
SIGNAL.SHUNT ":Signal -> Shaig"
SIG.FUNC ":Signal -> (num -> MAspect) @ ((num -> bool) @
  (num -> SubAspect)) + (num -> ShAspect)"
ON        ":Signal -> (num -> bool)"
OFF       ":Signal -> (num -> bool)"
SIGNAL.FAULT ":Signal -> (num -> bool)"

```

Definitions

```

ShAspect.TY_DEF ⊢ 3rep. TYPE_DEFINITION (TRP (λv tl. (v = INL one) ∧
  (LENGTH tl = 0) ∨ (v = INR (INL one)) ∧ (LENGTH tl = 0) ∨
  (v = INR (INR one)) ∧ (LENGTH tl = 0))) rep
ShAspect.ISO_DEF ⊢ (∀a. ABS_ShAspect (REP_ShAspect a) = a) ∧
  (∀r. TRP (λv tl. (v = INL one) ∧ (LENGTH tl = 0) ∨
    (v = INR (INL one)) ∧ (LENGTH tl = 0) ∨ (v = INR (INR one)) ∧
    (LENGTH tl = 0)) r = (REP_ShAspect (ABS_ShAspect r) = r))
sh_on_DEF ⊢ sh_on = ABS_ShAspect (Node (INL one) [])
sh_off_DEF ⊢ sh_off = ABS_ShAspect (Node (INR (INL one) []))

```

$sh_faulty_DEF \vdash sh_faulty = ABS_ShAspect (Node (INR (INR one))) []]$
 $Shaig_TY_DEF \vdash \exists rep. TYPE_DEFINITION (TRP (\lambda v tl. (\exists f. v = f) \wedge (LENGTH tl = 0))) rep$
 $Shaig_ISO_DEF \vdash (\forall a. ABS_Shaig (REP_Shaig a) = a) \wedge (\forall r. TRP (\lambda v tl. (\exists f. v = f) \wedge (LENGTH tl = 0))) r = (REP_Shaig (ABS_Shaig r) = r))$
 $SHUNTSIG_DEF \vdash \forall f. SHUNTSIG f = ABS_Shaig (Node f [])$
 $SHUNT_FUNC_DEF \vdash \forall a. SHUNT.FUNC (SHUNTSIG a) = a$
 $SHUNT_ON_DEF \vdash \forall a t. SHUNT.ON (SHUNTSIG a) t = (a t = sh.on)$
 $SHUNT_OFF_DEF \vdash \forall a t. SHUNT.OFF (SHUNTSIG a) t = (a t = sh.off)$
 $SHUNT_FAULT_DEF \vdash \forall a t. SHUNT.FAULT (SHUNTSIG a) t = (a t = sh.faulty)$
 $SubAspect_TY_DEF \vdash \exists rep. TYPE_DEFINITION (TRP (\lambda v tl. (v = INL one) \wedge (LENGTH tl = 0) \vee (v = INR one) \wedge (LENGTH tl = 0))) rep$
 $SubAspect_ISO_DEF \vdash (\forall a. ABS_SubAspect (REP_SubAspect a) = a) \wedge (\forall r. TRP (\lambda v tl. (v = INL one) \wedge (LENGTH tl = 0) \vee (v = INR one) \wedge (LENGTH tl = 0))) r = (REP_SubAspect (ABS_SubAspect r) = r))$
 $sub_not_show_DEF \vdash sub_not_show = ABS_SubAspect (Node (INL one) [])$
 $sub_off_DEF \vdash sub_off = ABS_SubAspect (Node (INR one) [])$
 $Subsig_TY_DEF \vdash \exists rep. TYPE_DEFINITION (TRP (\lambda v tl. (\exists f. v = f) \wedge (LENGTH tl = 0))) rep$
 $Subsig_ISO_DEF \vdash (\forall a. ABS_Subsig (REP_Subsig a) = a) \wedge (\forall r. TRP (\lambda v tl. (\exists f. v = f) \wedge (LENGTH tl = 0))) r = (REP_Subsig (ABS_Subsig r) = r))$
 $SUBSIG_DEF \vdash \forall f. SUBSIG f = ABS_Subsig (Node f [])$
 $SUB_FUNC_DEF \vdash \forall a. SUB.FUNC (SUBSIG a) = a$
 $SUB_OFF_DEF \vdash \forall a t. SUB.OFF (SUBSIG a) t = (a t = sub.off)$
 $Jsig_TY_DEF \vdash \exists rep. TYPE_DEFINITION (TRP (\lambda v tl. (\exists f. v = f) \wedge (LENGTH tl = 0))) rep$
 $Jsig_ISO_DEF \vdash (\forall a. ABS_Jsig (REP_Jsig a) = a) \wedge (\forall r. TRP (\lambda v tl. (\exists f. v = f) \wedge (LENGTH tl = 0))) r = (REP_Jsig (ABS_Jsig r) = r))$
 $JSIG_DEF \vdash \forall f. JSIG f = ABS_Jsig (Node f [])$
 $J_FUNC_DEF \vdash \forall j. J.FUNC (JSIG j) = j$

```

MAAspect.TY_DEF  $\vdash \exists \text{rep. TYPE.DEFINITION} (\text{TRP} (\lambda u \text{ tl. } (v = \text{INL one}) \wedge$ 
  (LENGTH tl = 0)  $\vee (v = \text{INR} (\text{INL one})) \wedge (\text{LENGTH tl} = 0) \vee$ 
  (v = INR (INR (INL one)))  $\wedge (\text{LENGTH tl} = 0) \vee$ 
  (v = INR (INR (INR (INL one))))  $\wedge (\text{LENGTH tl} = 0) \vee$ 
  (v = INR (INR (INR (INR (INL one))))))  $\wedge (\text{LENGTH tl} = 0) \vee$ 
  (v = INR (INR (INR (INR (INR (INL one))))))  $\wedge (\text{LENGTH tl} = 0) \vee$ 
  (v = INR (INR (INR (INR (INR (INR one))))))  $\wedge$ 
  (LENGTH tl = 0))) rep

MAAspect.ISO_DEF  $\vdash (\forall a. \text{ABS.MAspect} (\text{REP.MAspect } a) = a) \wedge$ 
  ( $\forall r. \text{TRP} (\lambda u \text{ tl. } (v = \text{INL one}) \wedge (\text{LENGTH tl} = 0) \vee$ 
  (v = INR (INL one))  $\wedge (\text{LENGTH tl} = 0) \vee (v = \text{INR} (\text{INR} (\text{INL one}))) \wedge$ 
  (LENGTH tl = 0)  $\vee (v = \text{INR} (\text{INR} (\text{INR} (\text{INL one})))) \wedge$ 
  (LENGTH tl = 0)  $\vee (v = \text{INR} (\text{INR} (\text{INR} (\text{INR} (\text{INL one})))) \wedge$ 
  (LENGTH tl = 0)  $\vee (v = \text{INR} (\text{INR} (\text{INR} (\text{INR} (\text{INR} (\text{INL one})))) \wedge$ 
  (LENGTH tl = 0)  $\vee (v = \text{INR} (\text{INR} (\text{INR} (\text{INR} (\text{INR} (\text{INR one})))))) \wedge$ 
  (LENGTH tl = 0)) r = (\text{REP.MAspect} (\text{ABS.MAspect } r) = r))

green_DEF  $\vdash \text{green} = \text{ABS.MAspect} (\text{Node} (\text{INL one}) [])$ 
double_yellow_DEF  $\vdash \text{double\_yellow} = \text{ABS.MAspect} (\text{Node} (\text{INR} (\text{INL one})) [])$ 
yellow_DEF  $\vdash \text{yellow} = \text{ABS.MAspect} (\text{Node} (\text{INR} (\text{INR} (\text{INL one}))) [])$ 
red_DEF  $\vdash \text{red} = \text{ABS.MAspect} (\text{Node} (\text{INR} (\text{INR} (\text{INR} (\text{INL one})))) [])$ 
green_flash_DEF
 $\vdash \text{green\_flash} = \text{ABS.MAspect} (\text{Node} (\text{INR} (\text{INR} (\text{INR} (\text{INR} (\text{INL one})))) [])$ 
double_yellow_flash_DEF  $\vdash \text{double\_yellow\_flash} =$ 
   $\text{ABS.MAspect} (\text{Node} (\text{INR} (\text{INR} (\text{INR} (\text{INR} (\text{INR} (\text{INL one})))))) [])$ 
yellow_flash_DEF  $\vdash \text{yellow\_flash} =$ 
   $\text{ABS.MAspect} (\text{Node} (\text{INR} (\text{INR} (\text{INR} (\text{INR} (\text{INR} (\text{INL one})))))) [])$ 
faulty_aspect_DEF  $\vdash \text{faulty\_aspect} =$ 
   $\text{ABS.MAspect} (\text{Node} (\text{INR} (\text{INR} (\text{INR} (\text{INR} (\text{INR} (\text{INR one})))))) [])$ 
Mtype.TY_DEF  $\vdash \exists \text{rep. TYPE.DEFINITION} (\text{TRP} (\lambda u \text{ tl. } (v = \text{INL one}) \wedge$ 
  (LENGTH tl = 0)  $\vee (v = \text{INR} (\text{INL one})) \wedge (\text{LENGTH tl} = 0) \vee$ 
  (v = INR (INR (INL one)))  $\wedge (\text{LENGTH tl} = 0) \vee$ 
  (v = INR (INR (INR (INL one))))  $\wedge (\text{LENGTH tl} = 0) \vee$ 
  (v = INR (INR (INR (INR one))))  $\wedge (\text{LENGTH tl} = 0))) \text{rep}$ 

Mtype.ISO_DEF  $\vdash (\forall a. \text{ABS.Mtype} (\text{REP.Mtype } a) = a) \wedge$ 
  ( $\forall r. \text{TRP} (\lambda u \text{ tl. } (v = \text{INL one}) \wedge (\text{LENGTH tl} = 0) \vee$ 
  (v = INR (INL one))  $\wedge (\text{LENGTH tl} = 0) \vee (v = \text{INR} (\text{INR} (\text{INL one}))) \wedge$ 
  (LENGTH tl = 0)  $\vee (v = \text{INR} (\text{INR} (\text{INR} (\text{INL one})))) \wedge$ 
  (LENGTH tl = 0)  $\vee (v = \text{INR} (\text{INR} (\text{INR} (\text{INR one})))) \wedge$ 
  (LENGTH tl = 0)) r = (\text{REP.Mtype} (\text{ABS.Mtype } r) = r))

```

```

two_aspect_DEF  $\vdash$  two_aspect = ABS.Mtype (Node (INL one)) []
three_aspect_DEF  $\vdash$  three_aspect = ABS.Mtype (Node (INR (INL one)) [])
four_aspect_DEF  $\vdash$  four_aspect = ABS.Mtype (Node (INR (INR (INL one)) []))
two_repeat_DEF  $\vdash$  two_repeat = ABS.Mtype (Node (INR (INR (INR (INL one)) [])) [])
three_repeat_DEF
 $\vdash$  three_repeat = ABS.Mtype (Node (INR (INR (INR (INR one)) [])) [])
Maig_TY_DEF  $\vdash$   $\exists$  rep. TYPE_DEFINITION (TRP ( $\lambda v t l. (\exists M f. v = M.f) \wedge$ 
  (LENGTH  $tl = 0$ ))) rep
Maig_ISO_DEF
 $\vdash$  ( $\forall a. \text{ABS.Maig}(\text{REP.Maig } a) = a$ )  $\wedge$  ( $\forall r. \text{TRP}(\lambda v t l. (\exists M f. v = M.f) \wedge$ 
  (LENGTH  $tl = 0$ )))  $r = (\text{REP.Maig}(\text{ABS.Maig } r) = r)$ )
MSIG_DEF  $\vdash \forall M f. \text{MSIG } M f = \text{ABS.Maig}(\text{Node}(M, f))$ 
M_TYPE_DEF  $\vdash \forall \text{type } af. M\_TYPE(\text{MSIG type } af) = \text{type}$ 
M_FUNC_DEF  $\vdash \forall \text{type } af. M\_FUNC(\text{MSIG type } af) = af$ 
M_ASPECT_DEF  $\vdash \forall \text{type } af t. M\_ASPECT(\text{MSIG type } af) t = af t$ 
MAIN_ON_DEF  $\vdash \forall st. \text{MAIN\_ON } st = (M\_ASPECT st = \text{red})$ 
MAIN_FAULTY_DEF  $\vdash \forall st. \text{MAIN\_FAULTY } st = (M\_ASPECT st = \text{faulty\_aspect})$ 
MAIN_OFF_DEF  $\vdash \forall st. \text{MAIN\_OFF } st = \neg \text{MAIN\_ON } st \wedge \neg \text{MAIN\_FAULTY } st$ 
RED_DEF  $\vdash \forall st. \text{RED } st = (M\_ASPECT st = \text{red})$ 
YELLOW_DEF  $\vdash \forall st. \text{YELLOW } st = (M\_ASPECT st = \text{yellow})$ 
Signal_TY_DEF  $\vdash \exists$  rep. TYPE_DEFINITION (TRP ( $\lambda v t l. (\exists n M. v = \text{INL}(n, M)) \wedge$ 
  (LENGTH  $tl = 0$ )  $\vee (\exists n M J. v = \text{INR}(\text{INL}(n, M, J))) \wedge$ 
  (LENGTH  $tl = 0$ )  $\vee (\exists n M S'. v = \text{INR}(\text{INR}(\text{INL}(n, M, S')))) \wedge$ 
  (LENGTH  $tl = 0$ )  $\vee (\exists n M S' J. v = \text{INR}(\text{INR}(\text{INR}(\text{INL}(n, M, S', J)))) \wedge$ 
  (LENGTH  $tl = 0$ )  $\vee (\exists n S'. v = \text{INR}(\text{INR}(\text{INR}(\text{INR}(n, S'))))) \wedge$ 
  (LENGTH  $tl = 0$ ))) rep
Signal_ISO_DEF  $\vdash$  ( $\forall a. \text{ABS.Signal}(\text{REP.Signal } a) = a$ )  $\wedge$ 
  ( $\forall r. \text{TRP}(\lambda v t l. (\exists n M. v = \text{INL}(n, M)) \wedge (\text{LENGTH } tl = 0) \vee$ 
    ( $\exists n M J. v = \text{INR}(\text{INL}(n, M, J))) \wedge (\text{LENGTH } tl = 0) \vee$ 
    ( $\exists n M S'. v = \text{INR}(\text{INR}(\text{INL}(n, M, S')))) \wedge (\text{LENGTH } tl = 0) \vee$ 
    ( $\exists n M S' J. v = \text{INR}(\text{INR}(\text{INR}(\text{INL}(n, M, S', J)))) \wedge$ 
    (LENGTH  $tl = 0$ )  $\vee (\exists n S'. v = \text{INR}(\text{INR}(\text{INR}(\text{INR}(n, S'))))) \wedge$ 
    (LENGTH  $tl = 0$ )))  $r = (\text{REP.Signal}(\text{ABS.Signal } r) = r)$ )
SIGNALM_DEF  $\vdash \forall n M. \text{SIGNALM } n M = \text{ABS.Signal}(\text{Node}(\text{INL}(n, M)) [])$ 
SIGNALNJ_DEF  $\vdash \forall n M J. \text{SIGNALM } J n M J =$ 
  ABS.Signal(Node(INR(INL(n, M, J)) []))

```

$\text{SIGNALMS_DEF} \vdash \forall n M S'. \text{SIGNALMS } n M S' =$
 $\text{ABS_Signal (Node (INR (INR (INL (n, M, S'))))) []}$

$\text{SIGNALMSJ_DEF} \vdash \forall n M S' J. \text{SIGNALMSJ } n M S' J =$
 $\text{ABS_Signal (Node (INR (INR (INL (n, M, S', J))))) []}$

$\text{SIGNALS_DEF} \vdash \forall n S'. \text{SIGNALS } n S' =$
 $\text{ABS_Signal (Node (INR (INR (INR (INR (n, S'))))) []}$

$\text{SIGNAL_ID_DEF} \vdash (\text{Vid } m. \text{SIGNAL_JD (SIGNALM id } m) = \text{id}) \wedge$
 $(\text{Vid } m j. \text{SIGNAL_JD (SIGNALMJ id } m j) = \text{id}) \wedge$
 $(\text{Vid } m s. \text{SIGNAL_JD (SIGNALMS id } m s) = \text{id}) \wedge$
 $(\text{Vid } m s j. \text{SIGNAL_JD (SIGNALMSJ id } m s j) = \text{id}) \wedge$
 $(\text{Vid } a h. \text{SIGNAL_JD (SIGNALS id } a h) = \text{id})$

$\text{SIGNAL_MAIN_DEF} \vdash (\text{Vid } m. \text{SIGNAL_MAIN (SIGNALM id } m) = m) \wedge$
 $(\text{Vid } m j. \text{SIGNAL_MAIN (SIGNALMJ id } m j) = m) \wedge$
 $(\text{Vid } m s. \text{SIGNAL_MAIN (SIGNALMS id } m s) = m) \wedge$
 $(\text{Vid } m s j. \text{SIGNAL_MAIN (SIGNALMSJ id } m s j) = m)$

$\text{SIGNAL_JUNC_DEF} \vdash (\text{Vid } m j. \text{SIGNAL_JUNC (SIGNALMJ id } m j) = j) \wedge$
 $(\text{Vid } m s j. \text{SIGNAL_JUNC (SIGNALMSJ id } m s j) = j)$

$\text{SIGNAL_SUB_DEF} \vdash (\text{Vid } m s. \text{SIGNAL_SUB (SIGNALMS id } m s) = s) \wedge$
 $(\text{Vid } m s j. \text{SIGNAL_SUB (SIGNALMSJ id } m s j) = s)$

$\text{SIGNAL_SHUNT_DEF} \vdash \text{Vid } a h. \text{SIGNAL_SHUNT (SIGNALS id } a h) = a h$

SIG_SFUNC_DEF
 $\vdash (\text{Vid } m. \text{SIG_SFUNC (SIGNALM id } m) = \text{INL (M.FUNC } m, \text{ARB. ARB)}) \wedge$
 $(\text{Vid } m j. \text{SIG_SFUNC (SIGNALMJ id } m j) =$
 $\text{INL (M.FUNC } m, \text{J.FUNC } j, \text{ARB)}) \wedge$
 $(\text{Vid } m s. \text{SIG_SFUNC (SIGNALMS id } m s) =$
 $\text{INL (M.FUNC } m, \text{ARB. SUB.FUNC } s) \wedge$
 $(\text{Vid } m s j. \text{SIG_SFUNC (SIGNALMSJ id } m s j) =$
 $\text{INL (M.FUNC } m, \text{J.FUNC } j, \text{SUB.FUNC } s) \wedge$
 $(\text{Vid } a h. \text{SIG_SFUNC (SIGNALS id } a h) = \text{INR (SHUNT.FUNC } a h))$

$\text{ON_DEF} \vdash (\text{Vid } m t. \text{ON (SIGNALM id } m) t = \text{MAIN.ON } m t) \wedge$
 $(\text{Vid } m j t. \text{ON (SIGNALMJ id } m j) t = \text{MAIN.ON } m t) \wedge$
 $(\text{Vid } m s t. \text{ON (SIGNALMS id } m s) t = \text{MAIN.ON } m t) \wedge$
 $(\text{Vid } m s j t. \text{ON (SIGNALMSJ id } m s j) t = \text{MAIN.ON } m t) \wedge$
 $(\text{Vid } a h t. \text{ON (SIGNALS id } a h) t = \text{SHUNT.ON } a h t)$

$\text{OFF_DEF} \vdash (\text{Vid } m t. \text{OFF (SIGNALM id } m) t = \text{MAIN.OFF } m t) \wedge$
 $(\text{Vid } m j t. \text{OFF (SIGNALMJ id } m j) t = \text{MAIN.OFF } m t) \wedge$
 $(\text{Vid } m s t. \text{OFF (SIGNALMS id } m s) t = \text{MAIN.OFF } m t) \wedge$
 $(\text{Vid } m s j t. \text{OFF (SIGNALMSJ id } m s j) t = \text{MAIN.OFF } m t) \wedge$
 $(\text{Vid } a h t. \text{OFF (SIGNALS id } a h) t = \text{SHUNT.OFF } a h t)$

$\text{SIGNAL_FAULT_DEF} \vdash \forall s t. \text{SIGNAL.FAULT } s t = \neg (\text{ON } s t \vee \text{OFF } s t)$

Theorems

ShAspect.Axiom $\vdash \forall e_0 e_1 e_2. (\exists \forall fn. (fn.sh_on = e_0) \wedge (fn.sh_off = e_1) \wedge (fn.sh_faulty = e_2))$

ShAspect.const.dist

$\vdash \neg(sh_on = sh_off) \wedge \neg(sh_on = sh_faulty) \wedge \neg(sh_off = sh_faulty)$

ShAspect.INDUCT $\vdash \forall P. P.sh_on \wedge P.sh_off \wedge P.sh_faulty \supset (\forall S'. P S')$

ShAspect.cases $\vdash \forall S'. (S' = sh_on) \vee (S' = sh_off) \vee (S' = sh_faulty)$

Shsig.Axiom $\vdash \forall f. (\exists \forall fn. (\forall f'. fn(SHUNTSIG f') = f f'))$

Shsig.one.one $\vdash \forall f' f''. (SHUNTSIG f' = SHUNTSIG f'') \Rightarrow (f' = f'')$

Shsig.INDUCT $\vdash \forall P. (\forall f'. P(SHUNTSIG f')) \supset (\forall S'. P S')$

Shsig.cases $\vdash \forall S'. (\exists f'. S' = SHUNTSIG f')$

SubAspect.Axiom $\vdash \forall e_0 e_1. (\exists \forall fn. (fn.sub_not_show = e_0) \wedge (fn.sub_off = e_1))$

SubAspect.const.dist $\vdash \neg(sub_not_show = sub_off)$

SubAspect.INDUCT $\vdash \forall P. P.sub_not_show \wedge P.sub_off \supset (\forall S'. P S')$

SubAspect.cases $\vdash \forall S'. (S' = sub_not_show) \vee (S' = sub_off)$

Subsig.Axiom $\vdash \forall f. (\exists \forall fn. (\forall f'. fn(SUBSIG f') = f f'))$

Subsig.one.one $\vdash \forall f' f''. (SUBSIG f' = SUBSIG f'') \Rightarrow (f' = f'')$

Subsig.INDUCT $\vdash \forall P. (\forall f'. P(SUBSIG f')) \supset (\forall S'. P S')$

Subsig.cases $\vdash \forall S'. (\exists f'. S' = SUBSIG f')$

Jsig.Axiom $\vdash \forall f. (\exists \forall fn. (\forall f'. fn(JSIG f') = f f'))$

Jsig.one.one $\vdash \forall f' f''. (JSIG f' = JSIG f'') \Rightarrow (f' = f'')$

Jsig.INDUCT $\vdash \forall P. (\forall f'. P(JSIG f')) \supset (\forall J. P J)$

Jsig.cases $\vdash \forall J. (\exists f'. J = JSIG f')$

MAAspect.Axiom $\vdash \forall e_0 e_1 e_2 e_3 e_4 e_5 e_6 e_7. (\exists \forall fn. (fn.green = e_0) \wedge (fn.double_yellow = e_1) \wedge (fn.yellow = e_2) \wedge (fn.red = e_3) \wedge (fn.green_flash = e_4) \wedge (fn.double_yellow_flash = e_5) \wedge (fn.yellow_flash = e_6) \wedge (fn.faulty_aspect = e_7))$

MAAspect.const.dist $\vdash \neg(green = double_yellow) \wedge \neg(green = yellow) \wedge$

$\neg(green = red) \wedge \neg(green = green_flash) \wedge$

$\neg(green = double_yellow_flash) \wedge \neg(green = yellow_flash) \wedge$

$\neg(green = faulty_aspect) \wedge \neg(double_yellow = yellow) \wedge$

$\neg(double_yellow = red) \wedge \neg(double_yellow = green_flash) \wedge$

$\neg(double_yellow = double_yellow_flash) \wedge$

$\neg(double_yellow = yellow_flash) \wedge \neg(double_yellow = faulty_aspect) \wedge$

$\neg(yellow = red) \wedge \neg(yellow = green_flash) \wedge$

$\neg(\text{yellow} = \text{double_yellow_flash}) \wedge \neg(\text{yellow} = \text{yellow_flash}) \wedge$
 $\neg(\text{yellow} = \text{faulty_aspect}) \wedge \neg(\text{red} = \text{green_flash}) \wedge$
 $\neg(\text{red} = \text{double_yellow_flash}) \wedge \neg(\text{red} = \text{yellow_flash}) \wedge$
 $\neg(\text{red} = \text{faulty_aspect}) \wedge \neg(\text{green_flash} = \text{double_yellow_flash}) \wedge$
 $\neg(\text{green_flash} = \text{yellow_flash}) \wedge \neg(\text{green_flash} = \text{faulty_aspect}) \wedge$
 $\neg(\text{double_yellow_flash} = \text{yellow_flash}) \wedge$
 $\neg(\text{double_yellow_flash} = \text{faulty_aspect}) \wedge \neg(\text{yellow_flash} = \text{faulty_aspect})$

$\text{MAspect_INDUCT} \vdash \forall P. P \text{ green} \wedge P \text{ double_yellow} \wedge P \text{ yellow} \wedge P \text{ red} \wedge P \text{ green_flash} \wedge$
 $P \text{ double_yellow_flash} \wedge P \text{ yellow_flash} \wedge P \text{ faulty_aspect} \supset (\forall M. P M)$

$\text{MAspect_cases} \vdash \forall M. (M = \text{green}) \vee (M = \text{double_yellow}) \vee (M = \text{yellow}) \vee$
 $(M = \text{red}) \vee (M = \text{green_flash}) \vee (M = \text{double_yellow_flash}) \vee$
 $(M = \text{yellow_flash}) \vee (M = \text{faulty_aspect})$

Mtype_Axiom
 $\vdash \forall e_0 e_1 e_2 e_3 e_4. (\exists \text{fn}. (\text{fn two_aspect} = e_0) \wedge (\text{fn three_aspect} = e_1) \wedge$
 $(\text{fn four_aspect} = e_2) \wedge (\text{fn two_repeat} = e_3) \wedge (\text{fn three_repeat} = e_4))$

$\text{Mtype_const_dist} \vdash \neg(\text{two_aspect} = \text{three_aspect}) \wedge \neg(\text{two_aspect} = \text{four_aspect}) \wedge$
 $\neg(\text{two_aspect} = \text{two_repeat}) \wedge \neg(\text{two_aspect} = \text{three_repeat}) \wedge$
 $\neg(\text{three_aspect} = \text{four_aspect}) \wedge \neg(\text{three_aspect} = \text{two_repeat}) \wedge$
 $\neg(\text{three_aspect} = \text{three_repeat}) \wedge \neg(\text{four_aspect} = \text{two_repeat}) \wedge$
 $\neg(\text{four_aspect} = \text{three_repeat}) \wedge \neg(\text{two_repeat} = \text{three_repeat})$

$\text{Mtype_INDUCT} \vdash \forall P. P \text{ two_aspect} \wedge P \text{ three_aspect} \wedge P \text{ four_aspect} \wedge P \text{ two_repeat} \wedge$
 $P \text{ three_repeat} \supset (\forall M. P M)$

$\text{Mtype_cases} \vdash \forall M. (M = \text{two_aspect}) \vee (M = \text{three_aspect}) \vee (M = \text{four_aspect}) \vee$
 $(M = \text{two_repeat}) \vee (M = \text{three_repeat})$

$\text{MSig_Axiom} \vdash \forall f. (\exists \text{fn}. (\forall M f'. \text{fn}(\text{MSIG } M f') = f M f'))$

$\text{MSig_one_one} \vdash \forall M f' M' f''. (\text{MSIG } M f' = \text{MSIG } M' f'') = (M = M') \wedge (f' = f'')$

$\text{MSig_INDUCT} \vdash \forall P. (\forall M f'. P(\text{MSIG } M f')) \supset (\forall M. P M)$

$\text{MSig_cases} \vdash \forall M. (\exists M' f'. M = \text{MSIG } M' f')$

$\text{Signal_Axiom} \vdash \forall f_0 f_1 f_2 f_3 f_4. (\exists \text{fn}. (\forall n M. \text{fn}(\text{SIGNALM } n M) = f_0 n M) \wedge$
 $(\forall n M J. \text{fn}(\text{SIGNALM } J n M) = f_1 n M J) \wedge$
 $(\forall n M S'. \text{fn}(\text{SIGNALMS } n M S') = f_2 n M S') \wedge$
 $(\forall n M S' J. \text{fn}(\text{SIGNALMS } J n M S') = f_3 n M S' J) \wedge$
 $(\forall n S'. \text{fn}(\text{SIGNALS } n S') = f_4 n S'))$

$\text{Signal_one_one} \vdash (\forall n M n' M'. (\text{SIGNALM } n M = \text{SIGNALM } n' M') =$
 $(n = n') \wedge (M = M')) \wedge$
 $(\forall n M J n' M' J'. (\text{SIGNALM } J n M = \text{SIGNALM } J n' M' J') =$
 $(n = n') \wedge (M = M') \wedge (J = J')) \wedge$
 $(\forall n M S' n' M' S''. (\text{SIGNALMS } n M S' = \text{SIGNALMS } n' M' S'') =$
 $(n = n') \wedge (M = M') \wedge (S' = S'')) \wedge$

$$\begin{aligned}
& (\forall n \ M \ S' \ J \ n' \ M' \ S'' \ J'. \ (\text{SIGNALMSJ} \ n \ M \ S' \ J = \\
& \quad \text{SIGNALMSJ} \ n' \ M' \ S'' \ J') = \\
& \quad (n = n') \wedge (M = M') \wedge (S' = S'') \wedge (J = J')) \wedge \\
& \quad (\forall n \ S' \ n' \ S''. \ (\text{SIGNALS} \ n \ S' = \text{SIGNALS} \ n' \ S'') = (n = n') \wedge (S' = S'')) \\
\text{Signal_INDUCT} \quad & \vdash \forall P. (\forall n \ M. \ P(\text{SIGNALM} \ n \ M)) \wedge \\
& \quad (\forall n \ M \ J. \ P(\text{SIGNALMJ} \ n \ M \ J)) \wedge (\forall n \ M \ S'. \ P(\text{SIGNALMS} \ n \ M \ S')) \wedge \\
& \quad (\forall n \ M \ S' \ J. \ P(\text{SIGNALMSJ} \ n \ M \ S' \ J)) \wedge \\
& \quad (\forall n \ S'. \ P(\text{SIGNALS} \ n \ S')) \supset (\forall S'. \ P \ S') \\
\text{Signal_cases} \quad & \vdash \forall S'. \ (\exists n \ M. \ S' = \text{SIGNALM} \ n \ M) \vee (\exists n \ M \ J. \ S' = \text{SIGNALMJ} \ n \ M \ J) \vee \\
& \quad (\exists n \ M \ S''. \ S' = \text{SIGNALMS} \ n \ M \ S'') \vee \\
& \quad (\exists n \ M \ S'' \ J. \ S' = \text{SIGNALMSJ} \ n \ M \ S'' \ J) \vee (\exists n \ S''. \ S' = \text{SIGNALS} \ n \ S'') \\
\text{SHUNT_NOT_ON_OFF} \quad & \vdash \forall st. \ \neg(\text{SHUNT.ON} \ st \wedge \text{SHUNT.OFF} \ st) \\
\text{SIGNAL_STATE} \quad & \vdash \forall st. \text{ON} \ st \vee \text{OFF} \ st \vee \text{SIGNAL.FAULT} \ st \\
\text{SIGNAL_NOT_ON_OFF} \quad & \vdash \forall st. \ \neg(\text{ON} \ st \wedge \text{OFF} \ st)
\end{aligned}$$

End of theory SIGNAL

A.6 The theory TRACK

Parents

SIGNAL

Types

":Ppos" ":Ploc" ":Point" ":Tstate" ":Tcir" ":Join"

Constants

```

REP.Ppos  ":Ppos -> (one + (one + one))ltree"
ABS.Ppos  ": (one + (one + one))ltree -> Ppos"
normal    ":Ppos"
reverse   ":Ppos"
moving    ":Ppos"
REP.Ploc  ":Ploc -> (one + (one + (one + one)))ltree"
ABS.Ploc  ": (one + (one + (one + one)))ltree -> Ploc"

```

```

free_move ":Ploc"
free_nor_rev ":Ploc"
free_rev_nor ":Ploc"
remote_locked ":Ploc"
REP_Point ":Point -> (num # ((num -> Ppos) # (num -> Ploc)))ltree"
ABS_Point ": (num # ((num -> Ppos) # (num -> Ploc)))ltree -> Point"
POINT "num -> ((num -> Ppos) -> ((num -> Ploc) -> Point))"
PNT_ID "Point -> num"
PNT_POS "Point -> (num -> Ppos)"
PNT_LOC "Point -> (num -> Ploc)"
PNT_LOCKED "Point -> (num -> bool)"
PNT_NORMAL "Point -> (num -> bool)"
PNT_REVERSE "Point -> (num -> bool)"
REP_Tstate ":Tstate -> (one + (one + one))ltree"
ABS_Tstate ": (one + (one + one))ltree -> Tstate"
occupied ":Tstate"
locked ":Tstate"
clear ":Tstate"
REP_Tcjr ":Tcjr -> (num # (num -> Tstate))ltree"
ABS_Tcjr ": (num # (num -> Tstate))ltree -> Tcjr"
TCJR "num -> ((num -> Tstate) -> Tcjr)"
TCJD "Tcjr -> num"
TC_SFUNC "Tcjr -> (num -> Tstate)"
TC_ST "Tcjr -> (num -> Tstate)"
TC_OCCUPIED "Tcjr -> (num -> bool)"
TC_CLEAR "Tcjr -> (num -> bool)"
TC_LOCKED "Tcjr -> (num -> bool)"
REP_Join ":Join -> (one + (one + (one + one)))ltree"
ABS_Join ": (one + (one + (one + one)))ltree -> Join"
J_conduct ":Join"
J_insulate ":Join"
J_overlap ":Join"

```

```

j.terminate ":Join"
IS_JOIN "":Join -> bool"
IS_JOIN "":Join -> bool"
IS_JOIN "":Join -> bool"
IS_JOIN "":Join -> bool"
IS_JOIN "":Join -> bool"

```

Definitions

```

Ppos.TY_DEF  $\vdash \exists rep. \text{TYPE.DEFINITION}(\text{TRP}(\lambda v tl. (v = \text{INL one}) \wedge$ 
   $(\text{LENGTH } tl = 0) \vee (v = \text{INR}(\text{INL one})) \wedge (\text{LENGTH } tl = 0) \vee$ 
   $(v = \text{INR}(\text{INR one})) \wedge (\text{LENGTH } tl = 0))) \text{ rep}$ 
Ppos.ISO_DEF  $\vdash (\forall a. \text{ABS.Ppos}(\text{REP.Ppos } a) = a) \wedge$ 
   $(\forall r. \text{TRP}(\lambda v tl. (v = \text{INL one}) \wedge (\text{LENGTH } tl = 0) \vee$ 
   $(v = \text{INR}(\text{INL one})) \wedge (\text{LENGTH } tl = 0) \vee (v = \text{INR}(\text{INR one})) \wedge$ 
   $(\text{LENGTH } tl = 0)) r = (\text{REP.Ppos}(\text{ABS.Ppos } r) = r))$ 
normal_DEF  $\vdash \text{normal} = \text{ABS.Ppos}(\text{Node}(\text{INL one})[])$ 
reverse_DEF  $\vdash \text{reverse} = \text{ABS.Ppos}(\text{Node}(\text{INR}(\text{INL one}))[])$ 
moving_DEF  $\vdash \text{moving} = \text{ABS.Ppos}(\text{Node}(\text{INR}(\text{INR one}))[])$ 
Ploc.TY_DEF  $\vdash \exists rep. \text{TYPE.DEFINITION}(\text{TRP}(\lambda v tl. (v = \text{INL one}) \wedge$ 
   $(\text{LENGTH } tl = 0) \vee (v = \text{INR}(\text{INL one})) \wedge (\text{LENGTH } tl = 0) \vee$ 
   $(v = \text{INR}(\text{INR}(\text{INL one}))) \wedge (\text{LENGTH } tl = 0) \vee$ 
   $(v = \text{INR}(\text{INR}(\text{INR one}))) \wedge (\text{LENGTH } tl = 0))) \text{ rep}$ 
Ploc.ISO_DEF  $\vdash (\forall a. \text{ABS.Ploc}(\text{REP.Ploc } a) = a) \wedge (\forall r. \text{TRP}(\lambda v tl. (v = \text{INL one}) \wedge$ 
   $(\text{LENGTH } tl = 0) \vee (v = \text{INR}(\text{INL one})) \wedge (\text{LENGTH } tl = 0) \vee$ 
   $(v = \text{INR}(\text{INR}(\text{INL one}))) \wedge (\text{LENGTH } tl = 0) \vee$ 
   $(v = \text{INR}(\text{INR}(\text{INR one}))) \wedge (\text{LENGTH } tl = 0)) r = (\text{REP.Ploc}(\text{ABS.Ploc } r) = r))$ 
free.move_DEF  $\vdash \text{free.move} = \text{ABS.Ploc}(\text{Node}(\text{INL one})[])$ 
free.nor_rev_DEF  $\vdash \text{free.nor.rev} = \text{ABS.Ploc}(\text{Node}(\text{INR}(\text{INL one}))[])$ 
free.rev.nor_DEF  $\vdash \text{free.rev.nor} = \text{ABS.Ploc}(\text{Node}(\text{INR}(\text{INR}(\text{INL one})))[])$ 
remote.locked_DEF  $\vdash \text{remote.locked} = \text{ABS.Ploc}(\text{Node}(\text{INR}(\text{INR}(\text{INR one})))[])$ 
Point.TY_DEF  $\vdash \exists rep. \text{TYPE.DEFINITION}(\text{TRP}(\lambda v tl. (\exists n f_0 f_1. v = n, f_0, f_1) \wedge$ 
   $(\text{LENGTH } tl = 0))) \text{ rep}$ 
Point.ISO_DEF  $\vdash (\forall a. \text{ABS.Point}(\text{REP.Point } a) = a) \wedge$ 
   $(\forall r. \text{TRP}(\lambda v tl. (\exists n f_0 f_1. v = n, f_0, f_1) \wedge$ 
   $(\text{LENGTH } tl = 0)) r = (\text{REP.Point}(\text{ABS.Point } r) = r))$ 
POINT_DEF  $\vdash \forall n f_0 f_1. \text{POINT } n f_0 f_1 = \text{ABS.Point}(\text{Node}(n, f_0, f_1)[])$ 

```

$\text{PNT_ID_DEF} \vdash \forall n \text{ pos loc. PNT_ID}(\text{POINT } n \text{ pos loc}) = n$
 $\text{PNT_POS_DEF} \vdash \forall n \text{ pos loc. PNT_POS}(\text{POINT } n \text{ pos loc}) = \text{pos}$
 $\text{PNT_LOC_DEF} \vdash \forall n \text{ pos loc. PNT_LOC}(\text{POINT } n \text{ pos loc}) = \text{loc}$
 $\text{PNT_LOCKED_DEF} \vdash \forall p t. \text{PNT_LOCKED } p t = (\text{PNT_LOC } p t = \text{remote.locked})$
 $\text{PNT_NORMAL_DEF} \vdash \forall p t. \text{PNT_NORMAL } p t = (\text{PNT_POS } p t = \text{normal})$
 $\text{PNT_REVERSE_DEF} \vdash \forall p t. \text{PNT_REVERSE } p t = (\text{PNT_POS } p t = \text{reverse})$
 $\text{Tstate_TY_DEF} \vdash \exists \text{rep. TYPE_DEFINITION}(\text{TRP}(\lambda v t l. (v = \text{INL one}) \wedge (\text{LENGTH } t l = 0) \vee (v = \text{INR}(\text{INL one})) \wedge (\text{LENGTH } t l = 0) \vee (v = \text{INR}(\text{INR one})) \wedge (\text{LENGTH } t l = 0)))) \text{rep}$
 $\text{Tstate_ISO_DEF} \vdash (\forall a. \text{ABS_Tstate}(\text{REP_Tstate } a) = a) \wedge (\forall r. \text{TRP}(\lambda v t l. (v = \text{INL one}) \wedge (\text{LENGTH } t l = 0) \vee (v = \text{INR}(\text{INL one})) \wedge (\text{LENGTH } t l = 0) \vee (v = \text{INR}(\text{INR one})) \wedge (\text{LENGTH } t l = 0)) r = (\text{REP_Tstate}(\text{ABS_Tstate } r) = r))$
 $\text{occupied_DEF} \vdash \text{occupied} = \text{ABS_Tstate}(\text{Node}(\text{INL one}))[]$
 $\text{locked_DEF} \vdash \text{locked} = \text{ABS_Tstate}(\text{Node}(\text{INR}(\text{INL one}))[])$
 $\text{clear_DEF} \vdash \text{clear} = \text{ABS_Tstate}(\text{Node}(\text{INR}(\text{INR one}))[])$
 $\text{Tcir_TY_DEF} \vdash \exists \text{rep. TYPE_DEFINITION}(\text{TRP}(\lambda v t l. (\exists n f. v = n, f) \wedge (\text{LENGTH } t l = 0)))) \text{rep}$
 $\text{Tcir_ISO_DEF} \vdash (\forall a. \text{ABS_Tcir}(\text{REP_Tcir } a) = a) \wedge (\forall r. \text{TRP}(\lambda v t l. (\exists n f. v = n, f) \wedge (\text{LENGTH } t l = 0)) r = (\text{REP_Tcir}(\text{ABS_Tcir } r) = r))$
 $\text{TCIR_DEF} \vdash \forall n f. \text{TCIR } n f = \text{ABS_Tcir}(\text{Node}(n, f))[]$
 $\text{TC_ID_DEF} \vdash \forall n s. \text{TC_ID}(\text{TCIR } n s) = n$
 $\text{TC_SFUNC_DEF} \vdash \forall n s. \text{TC_SFUNC}(\text{TCIR } n s) = s$
 $\text{TC_ST_DEF} \vdash \forall n s t. \text{TC_ST}(\text{TCIR } n s) t = s t$
 $\text{TC_OCCUPIED_DEF} \vdash \forall c t. \text{TC_OCCUPIED } c t = (\text{TC_ST } c t = \text{occupied})$
 $\text{TC_CLEAR_DEF} \vdash \forall c t. \text{TC_CLEAR } c t = (\text{TC_ST } c t = \text{clear})$
 $\text{TC_LOCKED_DEF} \vdash \forall c t. \text{TC_LOCKED } c t = (\text{TC_ST } c t = \text{locked})$
 $\text{Join_TY_DEF} \vdash \exists \text{rep. TYPE_DEFINITION}(\text{TRP}(\lambda v t l. (v = \text{INL one}) \wedge (\text{LENGTH } t l = 0) \vee (v = \text{INR}(\text{INL one})) \wedge (\text{LENGTH } t l = 0) \vee (v = \text{INR}(\text{INR}(\text{INL one}))) \wedge (\text{LENGTH } t l = 0) \vee (v = \text{INR}(\text{INR}(\text{INR one}))) \wedge (\text{LENGTH } t l = 0)))) \text{rep}$
 $\text{Join_ISO_DEF} \vdash (\forall a. \text{ABS_Join}(\text{REP_Join } a) = a) \wedge (\forall r. \text{TRP}(\lambda v t l. (v = \text{INL one}) \wedge (\text{LENGTH } t l = 0) \vee (v = \text{INR}(\text{INL one})) \wedge (\text{LENGTH } t l = 0) \vee (v = \text{INR}(\text{INR}(\text{INL one}))) \wedge (\text{LENGTH } t l = 0) \vee (v = \text{INR}(\text{INR}(\text{INR one}))) \wedge (\text{LENGTH } t l = 0)) r = (\text{REP_Join}(\text{ABS_Join } r) = r))$

$J_conduct_DEF \vdash J_conduct = ABS_Join (Node (INL one)) []$
 $J_insulate_DEF \vdash J_insulate = ABS_Join (Node (INR (INL one))) []$
 $J_overlap_DEF \vdash J_overlap = ABS_Join (Node (INR (INR (INL one)))) []$
 $J_terminate_DEF \vdash J_terminate = ABS_Join (Node (INR (INR (INR one)))) []$
 $IS_JCOND_DEF \vdash \forall j. IS_JCOND j = (j = J_conduct)$
 $IS_JINSU_DEF \vdash \forall j. IS_JINSU j = (j = J_insulate)$
 $IS_JOVER_DEF \vdash \forall j. IS_JOVER j = (j = J_overlap)$
 $IS_JTERM_DEF \vdash \forall j. IS_JTERM j = (j = J_terminate)$

Theorems

$Ppos_Axiom \vdash \forall e_0 e_1 e_2. (\exists f n. (f n \text{ normal} = e_0) \wedge (f n \text{ reverse} = e_1) \wedge$
 $(f n \text{ moving} = e_2))$
 $Ppos_const_dist$
 $\vdash \neg(\text{normal} = \text{reverse}) \wedge \neg(\text{normal} = \text{moving}) \wedge \neg(\text{reverse} = \text{moving})$
 $Ppos_INDUCT \vdash \forall P. P \text{ normal} \wedge P \text{ reverse} \wedge P \text{ moving} \supset (\forall P'. P P')$
 $Ppos_cases \vdash \forall P'. (P' = \text{normal}) \vee (P' = \text{reverse}) \vee (P' = \text{moving})$
 $Ploc_Axiom \vdash \forall e_0 e_1 e_2 e_3. (\exists f n. (f n \text{ free_move} = e_0) \wedge (f n \text{ free_nor_rev} = e_1) \wedge$
 $(f n \text{ free_rev_nor} = e_2) \wedge (f n \text{ remote_locked} = e_3))$
 $Ploc_const_dist \vdash \neg(\text{free_move} = \text{free_nor_rev}) \wedge \neg(\text{free_move} = \text{free_rev_nor}) \wedge$
 $\neg(\text{free_move} = \text{remote_locked}) \wedge \neg(\text{free_nor_rev} = \text{free_rev_nor}) \wedge$
 $\neg(\text{free_nor_rev} = \text{remote_locked}) \wedge \neg(\text{free_rev_nor} = \text{remote_locked})$
 $Ploc_INDUCT \vdash \forall P. P \text{ free_move} \wedge P \text{ free_nor_rev} \wedge P \text{ free_rev_nor} \wedge$
 $P \text{ remote_locked} \supset (\forall P'. P P')$
 $Ploc_cases \vdash \forall P'. (P' = \text{free_move}) \vee (P' = \text{free_nor_rev}) \vee (P' = \text{free_rev_nor}) \vee$
 $(P' = \text{remote_locked})$
 $Point_Axiom \vdash \forall f. (\exists f n. (\forall n f_0 f_1. f n (\text{POINT } n f_0 f_1) = f n f_0 f_1))$
 $Point_one_one \vdash \forall n f_0 f_1 n' f'_0 f'_1. (\text{POINT } n f_0 f_1 = \text{POINT } n' f'_0 f'_1) = (n = n') \wedge$
 $(f_0 = f'_0) \wedge (f_1 = f'_1)$
 $Point_INDUCT \vdash \forall P. (\forall n f_0 f_1. P (\text{POINT } n f_0 f_1)) \supset (\forall P'. P P')$
 $Point_cases \vdash \forall P'. (\exists n f_0 f_1. P' = \text{POINT } n f_0 f_1)$
 $Tstate_Axiom \vdash \forall e_0 e_1 e_2. (\exists f n. (f n \text{ occupied} = e_0) \wedge (f n \text{ locked} = e_1) \wedge$
 $(f n \text{ clear} = e_2))$
 $Tstate_const_dist$
 $\vdash \neg(\text{occupied} = \text{locked}) \wedge \neg(\text{occupied} = \text{clear}) \wedge \neg(\text{locked} = \text{clear})$

$\text{Tstate_INDUCT} \vdash \forall P. P \text{ occupied} \wedge P \text{ locked} \wedge P \text{ clear} \supset (\forall T'. P T')$
 $\text{Tstate_cases} \vdash \forall T'. (T' = \text{occupied}) \vee (T' = \text{locked}) \vee (T' = \text{clear})$
 $\text{Tcir_Axiom} \vdash \forall f. (\exists \forall f n. (\forall n f'. f n (\text{TCIR } n f') = f n f'))$
 $\text{Tcir_one_one} \vdash \forall n f' n' f''. (\text{TCIR } n f' = \text{TCIR } n' f'') = (n = n') \wedge (f' = f'')$
 $\text{Tcir_INDUCT} \vdash \forall P. (\forall n f'. P (\text{TCIR } n f')) \supset (\forall T'. P T')$
 $\text{Tcir_cases} \vdash \forall T'. (\exists n f'. T' = \text{TCIR } n f')$
 $\text{Join_Axiom} \vdash \forall e_0 e_1 e_2 e_3. (\exists \forall f n. (f n \text{ J.conduct} = e_0) \wedge (f n \text{ J.insulate} = e_1) \wedge$
 $(f n \text{ J.overlap} = e_2) \wedge (f n \text{ J.terminate} = e_3))$
 $\text{Join_const_dist} \vdash \neg(\text{J.conduct} = \text{J.insulate}) \wedge \neg(\text{J.conduct} = \text{J.overlap}) \wedge$
 $\neg(\text{J.conduct} = \text{J.terminate}) \wedge \neg(\text{J.insulate} = \text{J.overlap}) \wedge$
 $\neg(\text{J.insulate} = \text{J.terminate}) \wedge \neg(\text{J.overlap} = \text{J.terminate})$
 $\text{Join_INDUCT} \vdash \forall P. P \text{ J.conduct} \wedge P \text{ J.insulate} \wedge P \text{ J.overlap} \wedge P \text{ J.terminate} \supset (\forall J. P J)$
 $\text{Join_cases} \vdash \forall J. (J = \text{J.conduct}) \vee (J = \text{J.insulate}) \vee (J = \text{J.overlap}) \vee$
 $(J = \text{J.terminate})$

End of theory TRACK

A.7 The theory PART

Parents

TRACK SIGNAL

Types

"Part" "Elbl"

Constants

$\text{REP.Part} \text{ " :Part} \rightarrow (\text{num} + (\text{num} \# \text{Tcir} +$
 $(\text{num} \# (\text{Tcir} \# ((\text{num} \# \text{num}) \# (\text{num} \# \text{num}))) +$
 $\text{num} \# (\text{Tcir} \# (\text{Point} \# (\text{num} \# (\text{num} \# \text{num})))))) \text{ ltree}"$
 $\text{ABS.Part} \text{ " : (num} + (\text{num} \# \text{Tcir} + (\text{num} \# (\text{Tcir} \#$
 $((\text{num} \# \text{num}) \# (\text{num} \# \text{num}))) + \text{num} \# (\text{Tcir} \# (\text{Point} \#$
 $(\text{num} \# (\text{num} \# \text{num})))))) \text{ ltree} \rightarrow \text{Part}"$
 $\text{BPART} \text{ " : num} \rightarrow \text{Part}"$

```

TPART    ":num -> (Tcir -> Part)"
DPART    ":num -> (Tcir -> (num # num -> (num # num -> Part)))"
PPART    ":num -> (Tcir -> (Point -> (num # (num # num) -> Part)))"
PART.ID   ":Part -> num"
PART.CIRCUIT ":Part -> Tcir"
PART.POINT ":Part -> Point"
PART.PNT.TRAILING ":Part -> num"
PART.PNT.NORMAL ":Part -> num"
PART.PNT.REVERSE ":Part -> num"
PART.DIA1 ":Part -> num # num"
PART.DIA2 ":Part -> num # num"
IS.BPART  ":Part -> bool"
IS.TPART  ":Part -> bool"
IS.DPART  ":Part -> bool"
IS.PPART  ":Part -> bool"
REP.Elbl  ":Elbl -> (Join # Signal + Join)ltree"
ABS.Elbl  ": (Join # Signal + Join)ltree -> Elbl"
ELBLSIG   ":Join -> (Signal -> Elbl)"
ELBL      ":Join -> Elbl"
IS.ELBL.SIGNAL ":Elbl -> bool"
ELBL.SIGNAL ":Elbl -> Signal"
ELBL JOIN ":Elbl -> Join"

```

Definitions

```

Part.TY_DEF ⊢ 3rep. TYPE_DEFINITION (TRP (λv tl. (∃n. v = INL n) ∧
  (LENGTH tl = 0) ∨ (∃n T'. v = INR (INL (n, T')))) ∧ (LENGTH tl = 0) ∨
  (∃n T' p0 p1. v = INR (INR (INL (n, T', p0, p1)))) ∧ (LENGTH tl = 0) ∨
  (∃n T' P p. v = INR (INR (INR (n, T', P, p)))))) ∧ (LENGTH tl = 0))) rep

```

```

Part.ISO_DEF
  ⊢ (∀a. ABS.Part (REP.Part a) = a) ∧ (∀r. TRP (λv tl. (∃n. v = INL n) ∧
    (LENGTH tl = 0) ∨ (∃n T'. v = INR (INL (n, T')))) ∧ (LENGTH tl = 0) ∨
    (∃n T' p0 p1. v = INR (INR (INL (n, T', p0, p1)))) ∧ (LENGTH tl = 0) ∨
    (∃n T' P p. v = INR (INR (INR (n, T', P, p)))))) ∧
    (LENGTH tl = 0)) r = (REP.Part (ABS.Part r) = r))

```


$\text{BPART_DEF} \vdash \forall n. \text{BPART } n = \text{ABS.Part}(\text{Node}(\text{INL } n))[]$
 $\text{TPART_DEF} \vdash \forall n \, T'. \text{TPART } n \, T' = \text{ABS.Part}(\text{Node}(\text{INR}(\text{INL}(n, T'))))[]$
 $\text{DPART_DEF} \vdash \forall n \, T' \, p_0 \, p_1. \text{DPART } n \, T' \, p_0 \, p_1 =$
 $\text{ABS.Part}(\text{Node}(\text{INR}(\text{INR}(\text{INL}(n, T', p_0, p_1))))[])$
 $\text{PPART_DEF} \vdash \forall n \, T' \, P \, p. \text{PPART } n \, T' \, P \, p =$
 $\text{ABS.Part}(\text{Node}(\text{INR}(\text{INR}(\text{INR}(n, T', P, p))))[])$
 $\text{PART_ID_DEF} \vdash (\forall n. \text{PART_ID}(\text{BPART } n) = n) \wedge (\forall n \, t. \text{PART_ID}(\text{TPART } n \, t) = n) \wedge$
 $(\forall n \, t \, n_1 \, n_2. \text{PART_ID}(\text{DPART } n \, t \, n_1 \, n_2) = n) \wedge$
 $(\forall n \, t \, p \, n_3. \text{PART_ID}(\text{PPART } n \, t \, p \, n_3) = n)$
 $\text{PART_CIRCUIT_DEF} \vdash (\forall n \, t \, c. \text{PART_CIRCUIT}(\text{TPART } n \, t \, c) = t \, c) \wedge$
 $(\forall n \, t \, c \, n_1 \, n_2. \text{PART_CIRCUIT}(\text{DPART } n \, t \, c \, n_1 \, n_2) = t \, c) \wedge$
 $(\forall n \, t \, c \, p \, n_3. \text{PART_CIRCUIT}(\text{PPART } n \, t \, c \, p \, n_3) = t \, c)$
 $\text{PART_POINT_DEF} \vdash \forall n \, t \, c \, p \, n_3. \text{PART_POINT}(\text{PPART } n \, t \, c \, p \, n_3) = p$
 $\text{PART_PNT_TRAILING_DEF} \vdash \forall n \, t \, c \, p \, n_3. \text{PART_PNT_TRAILING}(\text{PPART } n \, t \, c \, p \, n_3) = \text{FST } n_3$
 $\text{PART_PNT_NORMAL_DEF} \vdash \forall n \, t \, c \, p \, n_3. \text{PART_PNT_NORMAL}(\text{PPART } n \, t \, c \, p \, n_3) = \text{FST}(\text{SND } n_3)$
 $\text{PART_PNT_REVERSE_DEF} \vdash \forall n \, t \, c \, p \, n_3. \text{PART_PNT_REVERSE}(\text{PPART } n \, t \, c \, p \, n_3) = \text{SND}(\text{SND } n_3)$
 $\text{PART_DIA_DEF} \vdash \forall n \, t \, c \, n_1 \, n_2. \text{PART_DIA1}(\text{DPART } n \, t \, c \, n_1 \, n_2) = n_1$
 $\text{PART_DIA2_DEF} \vdash \forall n \, t \, c \, n_1 \, n_2. \text{PART_DIA2}(\text{DPART } n \, t \, c \, n_1 \, n_2) = n_2$
 $\text{IS_BPART_DEF} \vdash (\forall n. \text{IS_BPART}(\text{BPART } n) = \text{T}) \wedge (\forall n \, t. \text{IS_BPART}(\text{TPART } n \, t) = \text{F}) \wedge$
 $(\forall n \, t \, n_1 \, n_2. \text{IS_BPART}(\text{DPART } n \, t \, n_1 \, n_2) = \text{F}) \wedge$
 $(\forall n \, t \, p \, n_3. \text{IS_BPART}(\text{PPART } n \, t \, p \, n_3) = \text{F})$
 $\text{IS_TPART_DEF} \vdash (\forall n. \text{IS_TPART}(\text{BPART } n) = \text{F}) \wedge (\forall n \, t. \text{IS_TPART}(\text{TPART } n \, t) = \text{T}) \wedge$
 $(\forall n \, t \, n_1 \, n_2. \text{IS_TPART}(\text{DPART } n \, t \, n_1 \, n_2) = \text{F}) \wedge$
 $(\forall n \, t \, p \, n_3. \text{IS_TPART}(\text{PPART } n \, t \, p \, n_3) = \text{F})$
 $\text{IS_DPART_DEF} \vdash (\forall n. \text{IS_DPART}(\text{BPART } n) = \text{F}) \wedge (\forall n \, t. \text{IS_DPART}(\text{TPART } n \, t) = \text{F}) \wedge$
 $(\forall n \, t \, n_1 \, n_2. \text{IS_DPART}(\text{DPART } n \, t \, n_1 \, n_2) = \text{T}) \wedge$
 $(\forall n \, t \, p \, n_3. \text{IS_DPART}(\text{PPART } n \, t \, p \, n_3) = \text{F})$
 $\text{IS_PPART_DEF} \vdash (\forall n. \text{IS_PPART}(\text{BPART } n) = \text{F}) \wedge (\forall n \, t. \text{IS_PPART}(\text{TPART } n \, t) = \text{F}) \wedge$
 $(\forall n \, t \, n_1 \, n_2. \text{IS_PPART}(\text{DPART } n \, t \, n_1 \, n_2) = \text{F}) \wedge$
 $(\forall n \, t \, p \, n_3. \text{IS_PPART}(\text{PPART } n \, t \, p \, n_3) = \text{T})$

$\text{Elbl_TY_DEF} \vdash \exists \text{rep. TYPE_DEFINITION} (\text{TRP} (\lambda v t l. (\exists J S'. v = \text{INL} (J, S')) \wedge$
 $(\text{LENGTH } tl = 0) \vee (\exists J. v = \text{INR } J) \wedge (\text{LENGTH } tl = 0))) \text{ rep}$
 $\text{Elbl_ISO_DEF} \vdash (\forall a. \text{ABS_Elbl} (\text{REP_Elbl } a) = a) \wedge$
 $(\forall r. \text{TRP} (\lambda v t l. (\exists J S'. v = \text{INL} (J, S')) \wedge (\text{LENGTH } tl = 0) \vee$
 $(\exists J. v = \text{INR } J) \wedge (\text{LENGTH } tl = 0))) r = (\text{REP_Elbl} (\text{ABS_Elbl } r) = r))$
 $\text{ELBLSIG_DEF} \vdash \forall J S'. \text{ELBLSIG } J S' = \text{ABS_Elbl} (\text{Node} (\text{INL} (J, S')) [])$
 $\text{ELBL_DEF} \vdash \forall J. \text{ELBL } J = \text{ABS_Elbl} (\text{Node} (\text{INR } J) [])$
 $\text{IS_ELBL_SIGNAL_DEF} \vdash \forall j s. \text{IS_ELBL_SIGNAL} (\text{ELBLSIG } j s) = \text{T}$
 $\text{ELBL_SIGNAL_DEF} \vdash \forall j s. \text{ELBL_SIGNAL} (\text{ELBLSIG } j s) = s$
 ELBL_JOIN_DEF
 $\vdash (\forall j s. \text{ELBL_JOIN} (\text{ELBLSIG } j s) = j) \wedge (\forall j. \text{ELBL_JOIN} (\text{ELBL } j) = j)$

Theorems

$\text{Part_Axiom} \vdash \forall f_0 f_1 f_2 f_3. (\exists \forall f n. (\forall n. f n (\text{BPART } n) = f_0 n) \wedge$
 $(\forall n T'. f n (\text{TPART } n T') = f_1 n T') \wedge$
 $(\forall n T' p_0 p_1. f n (\text{DPART } n T' p_0 p_1) = f_2 n T' p_0 p_1) \wedge$
 $(\forall n T' P p. f n (\text{PPART } n T' P p) = f_3 n T' P p))$
 $\text{Part_Induct} \vdash \forall P. (\forall n. P (\text{BPART } n)) \wedge (\forall n T'. P (\text{TPART } n T')) \wedge$
 $(\forall n T' p_0 p_1. P (\text{DPART } n T' p_0 p_1)) \wedge$
 $(\forall n T' P' p. P (\text{PPART } n T' P' p)) \supset (\forall P'. P P')$
 $\text{Part_one_one} \vdash (\forall n n'. (\text{BPART } n = \text{BPART } n') \Rightarrow (n = n')) \wedge$
 $(\forall n T' n' T''. (\text{TPART } n T' = \text{TPART } n' T'') \Rightarrow (n = n') \wedge (T' = T'')) \wedge$
 $(\forall n T' p_0 p_1 n' T'' p'_0 p'_1. (\text{DPART } n T' p_0 p_1 = \text{DPART } n' T'' p'_0 p'_1) \Rightarrow$
 $(n = n') \wedge (T' = T'') \wedge (p_0 = p'_0) \wedge (p_1 = p'_1)) \wedge$
 $(\forall n T' P p n' T'' P' p'. (\text{PPART } n T' P p = \text{PPART } n' T'' P' p') \Rightarrow$
 $(n = n') \wedge (T' = T'') \wedge (P = P') \wedge (p = p'))$
 $\text{Part_distinct} \vdash (\forall n n' T'. \neg (\text{BPART } n = \text{TPART } n' T')) \wedge$
 $(\forall n n' T' p_0 p_1. \neg (\text{BPART } n = \text{DPART } n' T' p_0 p_1)) \wedge$
 $(\forall n n' T' P p. \neg (\text{BPART } n = \text{PPART } n' T' P p)) \wedge$
 $(\forall n T' n' T'' p_0 p_1. \neg (\text{TPART } n T' = \text{DPART } n' T'' p_0 p_1)) \wedge$
 $(\forall n T' n' T'' P p. \neg (\text{TPART } n T' = \text{PPART } n' T'' P p)) \wedge$
 $(\forall n T' p_0 p_1 n' T'' P p. \neg (\text{DPART } n T' p_0 p_1 = \text{PPART } n' T'' P p))$
 $\text{Part_cases} \vdash \forall P'. (\exists n. P' = \text{BPART } n) \vee (\exists n T'. P' = \text{TPART } n T') \vee$
 $(\exists n T' p_0 p_1. P' = \text{DPART } n T' p_0 p_1) \vee$
 $(\exists n T' P' p. P' = \text{PPART } n T' P' p)$
 $\text{Elbl_Axiom} \vdash \forall f_0 f_1. (\exists \forall f n. (\forall J S'. f n (\text{ELBLSIG } J S') = f_0 J S') \wedge$
 $(\forall J. f n (\text{ELBL } J) = f_1 J))$
 $\text{Elbl_Induct} \vdash \forall P. (\forall J S'. P (\text{ELBLSIG } J S')) \wedge (\forall J. P (\text{ELBL } J)) \supset (\forall E. P E)$

$\text{Elbl_one_one} \vdash (\forall J S' J' S''. (\text{ELBLSIG } J S' = \text{ELBLSIG } J' S'') = (J = J') \wedge$
 $(S' = S'')) \wedge (\forall J J'. (\text{ELBL } J = \text{ELBL } J') = (J = J'))$
 $\text{Elbl_distinct} \vdash \forall J S' J'. \neg (\text{ELBLSIG } J S' = \text{ELBL } J')$
 $\text{Elbl_cases} \vdash \forall E. (\exists J S'. E = \text{ELBLSIG } J S') \vee (\exists J. E = \text{ELBL } J)$

End of theory PART

A.8 The theory NETWORK

Parents

HOL sets func graph alist path PART

Constants

$\text{NFC} \quad ":(\text{Part})\text{set} \Rightarrow (\text{Part} \Rightarrow (\text{Part} \Rightarrow \text{Elbl}))\text{set} \rightarrow (\text{Part} \rightarrow \text{bool})"$
 $\text{NJOIN} \quad ":(\text{Part})\text{set} \Rightarrow (\text{Part} \Rightarrow (\text{Part} \Rightarrow \text{Elbl}))\text{set} \rightarrow$
 $(\text{Part} \rightarrow (\text{Elbl} \rightarrow (\text{Part} \rightarrow (\text{Elbl} \rightarrow$
 $(\text{Part})\text{set} \Rightarrow (\text{Part} \Rightarrow (\text{Part} \Rightarrow \text{Elbl}))\text{set})))"$
 $\text{NETWORK} \quad ":(\text{Part})\text{set} \Rightarrow (\text{Part} \Rightarrow (\text{Part} \Rightarrow \text{Elbl}))\text{set} \rightarrow \text{bool}"$

Definitions

$\text{NFC_DEF} \quad \vdash (\forall n. \text{NFC } N (\text{BPART } n) = \text{IN_DEGREE } N (\text{BPART } n) < 1) \wedge$
 $(\forall n \text{ nt}. \text{NFC } N (\text{TPART } nt) = \text{IN_DEGREE } N (\text{TPART } nt) < 2) \wedge$
 $(\forall n \text{ nt } P n_3. \text{NFC } N (\text{PPART } nt P n_3) =$
 $\text{IN_DEGREE } N (\text{PPART } nt P n_3) < 3) \wedge (\forall n \text{ nt } n_1 n_2.$
 $\text{NFC } N (\text{DPART } nt n_1 n_2) = \text{IN_DEGREE } N (\text{DPART } nt n_1 n_2) < 4)$
 $\text{NJOIN_DEF} \quad \vdash \forall n_1 a_1 n_2 a_2. \text{NJOIN } N n_1 a_1 n_2 a_2 = (n_1, n_2, a_1) \text{INSERT_EDGE}$
 $((n_2, n_1, a_2) \text{INSERT_EDGE } (n_2 \text{INSERT_VERTEX } N))$
 $\text{NETWORK_DEF} \quad \vdash \forall N. \text{NETWORK } N = (\forall P. (\forall n. P(\{\{\text{BPART } n\}, \{\}\}) \wedge$
 $(\forall nt. P(\{\{\text{TPART } nt\}, \{\}\}) \wedge (\forall nt p n_3. P(\{\{\text{PPART } nt p n_3\}, \{\}\}) \wedge$
 $(\forall nt n_1 n_2. P(\{\{\text{DPART } nt n_1 n_2\}, \{\}\}) \wedge (\forall N' p_1 p_2. P N' \wedge$
 $\neg(p_1 = p_2) \wedge p_1 \text{IS_VERTEX } N' \wedge \text{NFC } N' p_1 \wedge$
 $\text{NFC } N' p_2 \supset (\forall a_1 a_2. P(\text{NJOIN } N' p_1 p_2 a_1 a_2))) \supset P N)$

Theorems

NETWORK_BUFFER $\vdash \forall n. \text{NETWORK}(\{\text{BPART } n\}, \{\})$
NETWORK_TRACK $\vdash \forall n \ell. \text{NETWORK}(\{\text{TPART } n \ell\}, \{\})$
NETWORK_POINT $\vdash \forall n \ell p n_3. \text{NETWORK}(\{\text{PPART } n \ell p n_3\}, \{\})$
NETWORK_DIAM $\vdash \forall n \ell n_1 n_2. \text{NETWORK}(\{\text{DPART } n \ell n_1 n_2\}, \{\})$
NETWORK_SIMP $\vdash \forall n. \text{NETWORK}(\{n\}, \{\})$
NETWORK_NJOIN $\vdash \forall N. \text{NETWORK } N \supset (\forall n_1 n_2. n_1 \text{ IS.VERTEX } N \wedge \neg(n_1 = n_2) \wedge$
 $\text{NFC } N \ n_1 \wedge \text{NFC } N \ n_2 \supset (\forall a_1 a_2. \text{NETWORK}(\text{NJOIN } N \ n_1 \ a_1 \ n_2 \ a_2)))$
NETWORK_INDUCT $\vdash \forall P. (\forall n. P(\{\text{BPART } n\}, \{\})) \wedge (\forall n \ell. P(\{\text{TPART } n \ell\}, \{\})) \wedge$
 $(\forall n \ell p n_3. P(\{\text{PPART } n \ell p n_3\}, \{\})) \wedge$
 $(\forall n \ell n_1 n_2. P(\{\text{DPART } n \ell n_1 n_2\}, \{\})) \wedge$
 $(\forall N \ p_1 \ p_2. P \ N \wedge \neg(p_1 = p_2) \wedge p_1 \text{ IS.VERTEX } N \wedge \text{NFC } N \ p_1 \wedge$
 $\text{NFC } N \ p_2 \supset (\forall a_1 a_2. P(\text{NJOIN } N \ p_1 \ a_1 \ p_2 \ a_2))) \supset$
 $(\forall N. \text{NETWORK } N \supset P \ N)$
NFC_SIMP $\vdash \forall n. \text{NFC}(\{n\}, \{\}) n$
NJOIN_EXP $\vdash \forall N \ n_1 \ a_1 \ n_2 \ a_2. n_1 \text{ IS.VERTEX } N \wedge \neg n_2 \text{ IS.VERTEX } N \supset$
 $(\text{NJOIN } N \ n_1 \ a_1 \ n_2 \ a_2 =$
 $n_2 \text{ INSERT VS } N, (n_1, n_2, a_1) \text{ INSERT } ((n_2, n_1, a_2) \text{ INSERT ES } N))$
NJOIN_EXP2 $\vdash \forall N \ n_1 \ a_1 \ n_2 \ a_2. n_1 \text{ IS.VERTEX } N \wedge n_2 \text{ IS.VERTEX } N \supset$
 $(\text{NJOIN } N \ n_1 \ a_1 \ n_2 \ a_2 =$
 $\text{VS } N, (n_1, n_2, a_1) \text{ INSERT } ((n_2, n_1, a_2) \text{ INSERT ES } N))$
NETWORK_GRAPH $\vdash \forall N. \text{NETWORK } N \supset \text{GRAPH } N$
NOT_VER_IMP_NFC $\vdash \forall N \ p. \text{NETWORK } N \supset (\neg p \text{ IS.VERTEX } N \supset \text{NFC } N \ p)$
NETWORK_FINITE $\vdash \forall N. \text{NETWORK } N \supset \text{FINITE}(\text{VS } N) \wedge \text{FINITE}(\text{ES } N)$
NETWORK_FINITE_GRAPH $\vdash \forall N. \text{NETWORK } N \supset \text{FINITE.GRAPH } N$
NETWORK_CONNECTED $\vdash \forall N. \text{NETWORK } N \supset \text{CONNECTED } N$

End of theory NETWORK

Appendix B

ML source listings

This appendix lists all the ML source files which create the theories in Appendix A. Each file is listed in a separate section.

B.1 The file mk_func.ml

```
new_theory'func';;
load_library'sets';;

new_special_symbol '~>>';;
new_special_symbol '~>-->';;
new_special_symbol '~<==>';;

let FUN_DEF = new_infdef_definition('FUN_DEF',
  "0<--> A B (f:a->b) = ((λ. (x IN A) => ((f x) IN B)))";;

let FUN_0UT0_DEF = new_infdef_definition('FUN_0UT0_DEF',
  "0<--> (λ:(α)out) (B:(set)out) f =
  ((λ. (x IN A) => (f x) IN B) /\
  (f y. (y IN A) => (y. (x IN A) /\ (y = f x))))";;

let FUN_0UT0_0UT0_DEF = new_infdef_definition('FUN_0UT0_0UT0_DEF',
  "0<--> (λ:(α)out) (B:(set)out) f =
  ((λ. (x IN A) => (f x) IN B) /\
  (λ y. (x IN A) /\ (y IN A) /\ ((f x) = (f y)) => (x = y)))";;

let FUN_100_DEF = new_infdef_definition('FUN_100_DEF',
  "0<--> (λ:(α)out) (B:(set)out) f =
  ((λ x--> B) f) /\ ((λ x--> B) f)";;

let FUN_0UT0_0 = prove_thm('FUN_0UT0_0',
  "(λ:(α)out) B C (f:a->b) (g:α->α) =
  ((λ x--> B) f) /\ ((λ x--> C) g) => ((λ x--> C) (g o f))",
  PURE_EXISTS_TAC[PURE_EXISTS_DEF:0_THM] THEN REPEAT 000_TAC THEN STRIP_TAC
  THEN 0001_TAC THEN 000_TAC THEN STRIP_TAC THEN 000_TAC THEN 000_TAC
```

```

THEN EXISTS_TAC "g" "a"
THEN SUBST1_TAC (ASSUME "y = (g:a → a) a")
THEN SUBST1_TAC (ASSUME "x = (f:a → a) a")
THEN CONJ_TAC THENL( FIRST_ASSUME ACCEPT_TAC; REFL_TAC);];

1st FOL_OBL_OBL_a = prove_thm('FOL_OBL_OBL_a',
  "!(A:(a)bool) B C (f:a→a) (g:a→a).
  ((A → B) ∧ (B → C)) ⇒ ((A → C) ∧ (g a f))",
  FOLM_REWRITE_TAC[FOL_OBL_OBL_a_THM] THEN REPEAT GEN_TAC THEN STRIP_TAC
  THEN CONJ_TAC THEN REPEAT GEN_TAC THEN STRIP_TAC THEN REFL_TAC THEN REFL_TAC);];

1st FOL_ISO_a = prove_thm('FOL_ISO_a',
  "!(A:(a)bool) B C (f:a→a) (g:a→a).
  ((A → B) ∧ (B → C)) ⇒ ((A → C) ∧ (g a f))",
  FOLM_REWRITE_TAC[FOL_ISO_OBL_a_THM] THEN REPEAT GEN_TAC THEN STRIP_TAC
  THEN INF_OBL_TAC FOL_OBL_OBL_a THEN INF_ASS_TAC FOL_OBL_OBL_a
  THEN CONJ_TAC THEN FIRST_ASSUME ACCEPT_TAC);];

1st FOL_INV_DEF = new_definition('FOL_INV_DEF',
  "FOL_INV (A:(a)bool) (B:(a)bool) ≡ y =
  ((y IN B) ∧ (x. (x IN A) ∧ (y = f x))) ⇒
  (Bx. (x IN A) ∧ (y = f x)) ∨ (Bx. x IN A)";];

1st FOL_INVERSO_DEF = new_definition('FOL_INVERSO_DEF',
  "FOL_INVERSO (A,B) (f:a→a) g =
  (A → B) ∧ (B → A) ∧ (f x. (x IN A) ⇒ ((g o f)x = x))";];

1st FOL_INVERSE_DEF = new_definition('FOL_INVERSE_DEF',
  "FOL_INVERSE (A,B) (f:a→a) g =
  (FOL_INVERSO (A,B) f) ∧ (FOL_INVERSO (B,A) g)";];

1st FOL_TT = prove_thm('FOL_TT',
  "!(A (f:a→a). (A → B) ⇒ (A → B) ⇒ ((A → B) ⇒
  FOLM_REWRITE_TAC[FOL_OBL_OBL_a_THM, FOL_INV_OBL, FOL_DEF]
  THEN REPEAT GEN_TAC THEN DISCH THEN STRIP_ASSUME_TAC
  THEN FIRST_ASSUME ACCEPT_TAC);];

1st FOL_INV_TT =
  let lam = REFLM_REWRITE_RULE[REFL_OBL, REFL_OBL, REFL_OBL] in
  prove_thm('FOL_INV_TT',
  "!(A (f:a→a). (A → B) ⇒ ((A → B) ⇒ (FOL_INV A B) ⇒
  REPEAT GEN_TAC THEN REWRITE_TAC[FOL_INV_DEF, lam]
  THEN DISCH_TAC THEN REPEAT STRIP_TAC THEN CONJ_TAC
  THENL[
    POP_ASSUME (\%. STRIP_ASSUME_TAC ((SELECT_BOL & CONJUNCT) %));
    FIRST_ASSUME (\%. MATCH_ACCEPT_TAC (SELECT_BOL %))];];);

1st LEFT_FOL = prove_thm('LEFT_FOL',
  "!(A B (f:a→a). (A → B) ∧ (A → B) ⇒
  FOL_INVERSO (A,B) f (FOL_INV A B) f",
  FOLM_REWRITE_TAC[FOL_INVERSO_DEF] THEN REPEAT GEN_TAC THEN STRIP_TAC
  THEN INF_ASS_TAC FOL_TT THEN INF_ASS_TAC FOL_INV_TT
  THEN REPEAT CONJ_TAC THEN (FIRST_ASSUME MATCH_ACCEPT_TAC ORLSS ALL_TAC)
  THEN FOLM_REWRITE_TAC[FOL_INV_DEF, a_THM]
  THEN UNDISCH_TAC "A → B (f:a→a)"
  THEN FOLM_REWRITE_TAC[FOL_OBL_OBL_a_THM]
  THEN REPEAT STRIP_TAC THEN REFL_TAC
```

```

THEN SURGEAL.THEN "(!n'. n' IN A /\ ((f:o-bool) x = f n'))" ASSUME_TAC
THEM1[
  EXISTS_TAC "n:o" THEN CONJ_TAC THEM1[FIRST_ASSUME ACCEPTY_TAC; DEFY_TAC];
  ASS_ASSUME_TAC1 THEN POP_ASSUME (\%. STRIP_ASSUME_TAC (SELECT_BNDR n))
  THEN RES_TAC THEN FIRST_ASSUME (\%. ACCEPTY_TAC (SVN n))];];

let RIGHT_PIV = prove.thm('RIGHT_PIV',
  "(!B (f:o-bool). ~(A = {}) /\ (A ==> B) f ==>
    FOL_PIVVERSE (B,A) (FOL_INV A B) f",
  FOL_ASSUME_TAC[FOL_PIVVERSE_DEF] THEN REPEAT RES_TAC THEN STRIP_TAC
  THEN IMP_RES_TAC FOL_PIV THEN IMP_RES_TAC FOL_INV THEN
  THEN REPEAT CONJ_TAC THEN (FIRST_ASSUME MATCH_ACCEPTY_TAC ORBLER ALL_TAC)
  THEN FOL_ASSUME_TAC[FOL_INV_DEF;o.THEM]
  THEN UNDISCH_TAC "(A ==> B) (f:o-bool)"
  THEN FOL_ORCH_ASSUME_TAC[FOL_OUTO_DEF]
  THEN REPEAT STRIP_TAC THEN RES_TAC
  THEN SURGEAL.THEN "(!n'. n' IN A /\ ((f:o-bool) x = f n'))" ASSUME_TAC
  THEM1[
    EXISTS_TAC "n:o" THEN CONJ_TAC THEN FIRST_ASSUME ACCEPTY_TAC;
    FILTER_ASS_ASSUME_TAC (\%. not(fact(strip.comb n) = "n:o-bool-bool"))
    THEN POP_ASSUME (\%. STRIP_ASSUME_TAC (SELECT_BNDR n))
    THEN FIRST_ASSUME (\%. MATCH_ACCEPTY_TAC (SVN n))];];

let LEFT_RIGHT_PIV = prove.thm('LEFT_RIGHT_PIV',
  "(!B (f:o-bool) g.
    (A ==> B) f /\ (B ==> A) g /\ FOL_PIVVERSE (A,B) f g ==>
    (A ==> B) f /\ (B ==> A) g",
  REPEAT RES_TAC THEN RESUME_TAC
    [FOL_DEF;FOL_ORCH_DEF;FOL_INV_DEF;FOL_PIVVERSE_DEF;o.THEM]
  THEN REPEAT STRIP_TAC THEN RES_TAC THEM1[
    UNDISCH_TAC "(g:o-bool)(f g) = g"
    THEN DISCH_THEN (\%. FUNSTY_TAC (SVN n))
    THEN UNDISCH_TAC "(g:o-bool)(f g) = n"
    THEN DISCH_THEN (\%. SUBSTY_TAC (SVN n))
    THEN AP_THM_TAC THEN FIRST_ASSUME ACCEPTY_TAC;
    EXISTS_TAC "(f:o-bool) g"
    THEN CONJ_TAC THEN (CONV_TAC SVN_CONV ORBLER ALL_TAC)
    THEN FIRST_ASSUME ACCEPTY_TAC];];

let ISO_INVERSE = prove.thm('ISO_INVERSE',
  "(!B (f:o-bool) g.
    (A ==> B) f /\ (B ==> A) g /\ FOL_INVERSE (A,B) f g ==>
    (A ==> B) f /\ (B ==> A) g",
  REPEAT RES_TAC THEN RESUME_TAC[FOL_INV_DEF;FOL_INVERSE_DEF]
  THEN REPEAT STRIP_TAC THEN IMP_RES_TAC LEFT_RIGHT_PIV];];

let FOL_EMPTY_LEFT =
  let loc1 = TAC_PROOF([], "(!B (f:o-bool). ((f ==> B) f-),
    RESUME_TAC[FOL_DEF;FOL_INV_DEF;FOL_EMPTY] in
  let loc2 = TAC_PROOF([], "(!B (f:o-bool). ((f ==> B) f-),
    RESUME_TAC[FOL_ORCH_DEF;FOL_INV_DEF;FOL_EMPTY] in
  let loc3 = TAC_PROOF([], "(!B (f:o-bool). (((f ==> B) (B = {})))",
    RESUME_TAC[FOL_OUTO_DEF;FOL_INV_EMPTY]
    THEN RES_TAC THEN CONV_TAC (ORCH_DEPTH_CONV FORALL_NOT_CONV)
    THEN RESUME_TAC[FORALL_NOT_EMPTY] in
  let loc4 = TAC_PROOF([], "(!B (f:o-bool). (((f ==> B) (B = {})))",
    REPEAT RES_TAC THEN

```

```

      REWRITE_TAC[PW_150_DWP;lem3;lem3] in
      save_thm('PW_EMPTY_LEFT', LIST_CONJ [lem1;lem2;lem3;lem4]);;

let PW_EMPTY_RIGHT =
  let lem = CONV_RULE (CONV_EMPTY_CONV SET_EMPTY_CONV)
    (GEN_ALL [PW_EMPTY_REWRITE_RULE[SET_EMPTY_CLASSES]
      (CONSTRAPOS(AND(IMP_NORM (SPEC_ALL MEMBERS_NOT_EMPTY))))]) in
  let lem1 = TAC_PROOF([[]], "!(a:(e->bool). (A --> {}))% (A = {})",
    REWRITE_TAC[PW_DWP;NOT_IN_EMPTY])
  THEN REPEAT GEN_TAC THEN CONV_TAC (CONV_EMPTY_CONV FORALL_NOT_CONV)
  THEN REWRITE_TAC[MEMBERS_NOT_EMPTY] in
  let lem2 = TAC_PROOF([[]], "!(a:(e->bool). (A --> {}))% (A = {})",
    REWRITE_TAC[PW_EMPTY_EMPTY_CONV;NOT_IN_EMPTY])
  THEN REPEAT GEN_TAC THEN EQ_TAC THEN STRIP_TAC
  THENL [IMP_RES_TAC lem;
    ASS_REWRITE_TAC[NOT_IN_EMPTY]] in
  let lem3 = TAC_PROOF([[]], "!(a:(e->bool). ((A --> {})% (A = {}))%3,
    REWRITE_TAC[PW_EMPTY_CONV;NOT_IN_EMPTY])
  THEN GEN_TAC THEN EQ_TAC
  THENL [MATCH_ACCEPT_TAC lem;
    DISCH_THEN (lambda SUBST1.TAC a) THEN REWRITE_TAC[NOT_IN_EMPTY]] in
  let lem4 = TAC_PROOF([[]], "!(a:(e->bool). ((A <--> {})% (A = {}))%3,
    REPEAT GEN_TAC THEN
      REWRITE_TAC[PW_150_DWP;lem3;lem3] in
      save_thm('PW_EMPTY_RIGHT', LIST_CONJ [lem1;lem2;lem3;lem4]);;

let PW_1 =
  let lem1 = TAC_PROOF([[]], "!(a:(e)set. (A --> A) ({}<->bool)",
    REWRITE_TAC[PW_DWP;I_THM]) in
  let lem2 = TAC_PROOF([[]], "!(a:(e)set. (A --> A) ({}<->bool)",
    REWRITE_TAC[PW_EMPTY_EMPTY_CONV;I_THM])
  THEN REPEAT STRIP_TAC THEN FIRST_ASSUM ACCEPT_TAC in
  let lem3 = TAC_PROOF([[]], "!(a:(e)set. (A --> A) ({}<->bool)",
    REWRITE_TAC[PW_EMPTY_CONV;I_THM])
  THEN REPEAT STRIP_TAC THEN REWRITE_TAC "p-a" THEN
    ASS_REWRITE_TAC[] in
  let lem4 = TAC_PROOF([[]], "!(a:(e)set. (A <--> A) ({}<->bool)",
    REWRITE_TAC[PW_150_DWP;I_THM])
  THEN REPEAT STRIP_TAC THENL [
    MATCH_ACCEPT_TAC lem3; MATCH_ACCEPT_TAC lem3] in
  save_thm('PW_1', LIST_CONJ [lem1;lem2;lem3;lem4]);;

let ISO_FIBV = save_thm('ISO_FIBV',
  "!(A B (e->bool). (A <--> B)% (B <--> A) (PW_150_A B B))",
  REPEAT GEN_TAC THEN REWRITE_TAC[PW_150_DWP]
  THEN ASS_CHOOSE_TAC "(A:(e)set) = {}"
  THEN ASS_REWRITE_TAC[PW_EMPTY_LEFT;PW_EMPTY_RIGHT]
  THEN ASS_CHOOSE_TAC "(B:(e)set) = {}"
  THEN ASS_REWRITE_TAC[PW_EMPTY_LEFT;PW_EMPTY_RIGHT]
  THEN REPEAT STRIP_TAC THEN MAP_FIRST IMP_RES_TAC
  [PW_TV;PW_150_TV;LEFT_FIBV;RIGHT_FIBV;LEFT_RIGHT_FIBV]
  THEN FIRST_ASSUM MATCH_MP_TAC THEN FIRST_ASSUM MATCH_ACCEPT_TAC);;

close_theory();;

```


B.2 The file mk_graph.ml

```

%-----
FILE: mk_graph.ml  ver:0.3
AUTHOR: Mai Kung DATE: 1 AUG 1990 modified Jun 91
%-----

new_theory 'graph';;

load_library 'sets';;

new_parent 'func';;
autoload_all 'func';;

set_flag('sticky', true);;

%-----
% General theorems and tactics needed in this theory -%
%-----

let PAIR_EQ_EQ = TAC_PROOF([],
  ~!(a:obs) g. (a = g) = (FUT a = FUT g) /\ (SND a = SND g)~,
  REPEAT QM_TAC
  THEN PURE_OBCH_REWRITE_TAC[QVM(SPEC ALL PAIR)]
  THEN PURE_OBCH_REWRITE_TAC[FUT;SND]
  THEN PURE_OBCH_REWRITE_TAC[PAIR_EQ]
  THEN PURE_OBCH_REWRITE_TAC[PAIR_EQ]
  THEN REPT_TAC;;

let PAIR_EQ_TAC = TAC_PROOF([],
  ~!(a:b) (b==) x y. ~!(a,b) = (a,y) ~ (a=x) /\ ~(b=y)~,
  REPEAT QM_TAC THEN EQ_TAC
  THEN QVM_TAC CONTRAPOSC_QVM
  THEN OBCH_REWRITE_TAC[QVM.NORMAN_THM]
  THEN OBCH_REWRITE_TAC[PAIR_EQ]
  THEN OBCH_REWRITE_TAC[];;

let SET_NULL_APPEND = TAC_PROOF([],
  ~!(l:o list) l2. 'NULL l2 ==> 'NULL (APPEND l l2)~,
  LIST_REDUCT_TAC THEN (REWRITE_TAC[APPEND;NULL]);;

let NULL_NIL = TAC_PROOF([],
  ~!(o)l1:l2. 'NULL l = (l = [])~,
  LIST_REDUCT_TAC THEN (REWRITE_TAC[NULL;SET_CONS_NIL]);;

let HD_APPEND = TAC_PROOF([],
  ~!p1 p2:(o list) ('NULL p1 ==> 'HD (APPEND p1 p2) = HD p1)~,
  LIST_REDUCT_TAC THEN [
    REWRITE_TAC[APPEND;NULL];
    OBCH_REWRITE_TAC[APPEND;NULL] THEN REWRITE_TAC[HD];;
  ];

let EVERY_APPEND = TAC_PROOF([],
  ~!(l:o list) l2 P. EVERY P (APPEND l l2) = (EVERY P l) /\ (EVERY P l2)~,
  LIST_REDUCT_TAC THEN [
    REWRITE_TAC[APPEND;EVERY_DEF];
    OBCH_REWRITE_TAC[APPEND;EVERY_DEF] THEN OBCH_REWRITE_TAC[EVERY_DEF]
    THEN ASH_REWRITE_TAC[] THEN REWRITE_TAC[CONJ_ASSOC];;
  ];

```

```

let DISJOINT_INSERT_TMP = TAC_PROOF([
  "(a:e) a v. DISJOINT (a INSERT a) s ==>
  (DISJOINT a v) /\ ~(a IN v)",
  MMWRITE_TAC(DISJOINT_DEF;INSERT;INVS)
  THEN REPEAT GEN_TAC THEN COND_CASE_TAC
  THEN MMWRITE_TAC(DISJOINT_INSERT;EMPTY);];

let IS_UNION_INSERT_TMP = TAC_PROOF([
  "(a:e) a v. (a IN v) ==> a IN (UNION a)",
  MMWRITE_TAC(IS_UNION;IS_UNION_INSERT);];

-----
% Vertices, Edge and Graph are defined as abbreviations for the types used
% to represent vertices, edges and graphs. %
%-----
let Vertices = "(a)" and
  Edge = ":(a -> a -> a)" and
  Graph = ":(a)set -> (a -> a -> a)set";;

-----
% The following three definitions are required by the definition of graph%
% s.src(a) is the source of an edge a %
% s.dest(a) is the destination of an edge a %
% s.lb(a) is the label of an edge a %
%-----
let s.src_DEF = new_definition('s.src_DEF',
  "s.src a := PST a");;

let s.dest_DEF = new_definition('s.dest_DEF',
  "s.dest a := PST (SND a)");;

let s.lb_DEF = new_definition('s.lb_DEF',
  "s.lb a := SBD (SBD a)");;

let s.src = prove_thm('s.src', "[p1 p2 a. s.src(p1,p2,a):'Edge' = p1",
  REPEAT GEN_TAC THEN PURS_ONCE_MMWRITE_TAC(s.src_DEF)
  THEN PURS_ONCE_MMWRITE_TAC[PST] THEN REFL_TAC);;

let s.dest = prove_thm('s.dest', "[p1 p2 a. s.dest(p1,p2,a):'Edge' = p2",
  REPEAT GEN_TAC THEN PURS_ONCE_MMWRITE_TAC(s.dest_DEF)
  THEN PURS_ONCE_MMWRITE_TAC[PST;SBD] THEN REFL_TAC);;

let s.lb = prove_thm('s.lb', "[p1 p2 a. s.lb(p1,p2,a):'Edge' = a",
  REPEAT GEN_TAC THEN PURS_ONCE_MMWRITE_TAC(s.lb_DEF)
  THEN PURS_ONCE_MMWRITE_TAC[PST;SBD] THEN REFL_TAC);;

-----
% A graph, by definition, is a pair of sets, where -%
% V is the vertex set, which can be a set of any type and -%
% E is the edge set which is a set of vertex pairs and labels. -%
% The constraint on graph is that all vertices appeared in -%
% the vertex pairs in the edge set are members of the vertex set. -%
%-----
let GRAPH_DEF = new_definition('GRAPH_DEF',
  "GRAPH ((V:('Vertices)),(E:('Edge)set)) =
  {a. a IN E ==> ((a.src a) IN V) /\ ((a.dest a) IN V)}");;

```

```

% A special graph is the empty graph %
let NULL_GRAPH = new_definition('NULL_GRAPH',
  "NULL_GRAPH = ((EMPTY('Vertex)set), (EMPTY('Edge)set))");

% The vertices set of a graph %
let VS_DEF = new_definition('VS_DEF', "VS (G:'Graph) = PUT 0");

% The edge set of a graph %
let ES_DEF = new_definition('ES_DEF', "ES (G:'Graph) = SET 0");

let VERIFIED = prove_thm('VERIFIED',
  "(V:('Vertex)set) (E:('Edge)set) VS(V,E) = V",
  REWRITE_TAC[VS_DEF;PUT]);

let EDGED = prove_thm('EDGED',
  "(V:('Vertex)set) (E:('Edge)set) ES(V,E) = E",
  REWRITE_TAC[ES_DEF;SET]);

% = IS_EDGE 0 iff a is in (ES G) %
let IS_EDGE_DEF = new_defn_definition('IS_EDGE_DEF',
  "IS_EDGE (G:'Graph) = a IS (ES G)");

% = IS_VERTEX 0 iff w is in (VS G) %
let IS_VERTEX_DEF = new_defn_definition('IS_VERTEX_DEF',
  "IS_VERTEX (G:'Graph) = w IS (VS G)");

%-----%
% Some basic facts about graph %
%-----%
% There exists a graph %
let GRAPH_EXISTS = prove_thm('GRAPH_EXISTS',
  "%(G:'Graph). GRAPH G",
  EXISTW_TAC["NULL_GRAPH" "Graph"
  THEN REWRITE_TAC[NULL_GRAPH;GRAPH_DEF;NOT_IS_EMPTY]]);

let GRAPH_PAIR = prove_thm('GRAPH_PAIR',
  "%(G:'Graph). GRAPH G ==> (E = (VS G, ES G))",
  REWRITE_TAC[VS_DEF;ES_DEF]);

let GRAPH_DECOMP = prove_thm('GRAPH_DECOMP',
  "%(G:'Graph). (GRAPH G) = (GRAPH (VS G, ES G))",
  REWRITE_TAC[VS_DEF;ES_DEF]);

let GRAPH_EQ = prove_thm('GRAPH_EQ',
  "%(G:'Graph) E. GRAPH G /\ GRAPH E ==>
  ((G = E) == ((VS G) = (VS E) /\ (ES G) = (ES E)))",
  REWRITE_TAC[VS_DEF;ES_DEF]
  THEN REPEAT STRIP_TAC THEN MATCH_ACCEPT_TAC GRAPH_PAIR_EQ);

let NOT_VERTEX_NOT_EDGE = prove_thm('NOT_VERTEX_NOT_EDGE',
  "%(G:'Graph) v1 v2 x. (GRAPH G) ==>
  ('v1 IS_VERTEX G /\ 'v2 IS_VERTEX G ==> ('v1,v2,x) IS_EDGE G",
  PUSH_ONCE REWRITE_TAC[GRAPH_DECOMP] THEN
  PUSH_REWRITE_TAC[GRAPH_DEF IS_EDGE_DEF IS_VERTEX_DEF]
  THEN REPEAT STRIP_TAC THEN CHV_TAC CONTRAPOS_CHV

```

```

THEN REWRITE_TAC[DE_NORMAL_THM] THEN DISCH_TAC THEN RES_THEN SP_TAC
THEN REWRITE_TAC[a_arc:a_def] THEN REPEAT STRIP_TAC
THEN FIRST_ASSUM ACCEPT_TAC::];

let GRAPH_BUT_VERTEX_BUT_EDGE = prove_thm('GRAPH_BUT_VERTEX_BUT_EDGE',
"((G:Graph) v. (GRAPH G) /\ ~(v IS_VERTEX G) ==>
  in x. ~((v,u,x) IS_EDGE G)",
  PUSH_BUCH_REWRITE_TAC[GRAPH_BUCHBSP]
  THEN REWRITE_TAC[GRAPH_DEF;IS_EDGE_DEF;IS_VERTEX_DEF]
  THEN REPEAT GEM_TAC THEN STRIP_TAC THEN REPEAT GEM_TAC
  THEN ASSUM_LIST (\a1. ASSUME_TAC
    (CONTRAPOS (ISPEC "(v,u,x):Edge" (a1 2 a1))))
  THEN POP_ASSUM (\v. ASSUME_TAC (REWRITE_RULE[DE_NORMAL_THM;a_def;a_arc] v))
  THEN RES_TAC::];

let GRAPH_BUT_VERTEX_BUT_EDGE2 = prove_thm('GRAPH_BUT_VERTEX_BUT_EDGE2',
"((G:Graph) v. (GRAPH G) /\ ~(v IS_VERTEX G) ==>
  in x. ~((u,v,x) IS_EDGE G)",
  PUSH_BUCH_REWRITE_TAC[GRAPH_BUCHBSP]
  THEN REWRITE_TAC[GRAPH_DEF;IS_EDGE_DEF;IS_VERTEX_DEF]
  THEN REPEAT GEM_TAC THEN STRIP_TAC THEN REPEAT GEM_TAC
  THEN ASSUM_LIST (\a1. ASSUME_TAC
    (CONTRAPOS (ISPEC "(u,v,x):Edge" (a1 2 a1))))
  THEN POP_ASSUM (\v. ASSUME_TAC (REWRITE_RULE[DE_NORMAL_THM;a_def;a_arc] v))
  THEN RES_TAC::];

-----
% Loops and multiple edges      -%
-----
% A loop is an edge having identical end points -%
let LOOP_DEF = new_definition('LOOP_DEF',
  "LOOP (e:Edge) = (a_arc e = a_def e)");

let HAS_LOOP_DEF = new_definition('HAS_LOOP_DEF',
  "HAS_LOOP G = ?(e:Edge). (e IS (EX G)) /\ (LOOP e)");

% Multiple edges are distinct edges but having the same end points -%
let MULTI_EDGE_DEF = new_definition('MULTI_EDGE_DEF',
  "MULTI_EDGE G = ?(e1:Edge) a1.
    (a1 IS (EX G)) /\ (a2 IS (EX G)) /\ ~(a1 = a2) /\
    (a_arc a1 = a_arc a2) /\ (a_def a1 = a_def a2)");

-----
% Simple graphs                -%
-----
% A simple graph is a graph without loop and without multiple edges %
let SIMPLE_GRAPH_DEF = new_definition('SIMPLE_GRAPH_DEF',
  "SIMPLE_GRAPH (G:Graph) =
    (GRAPH G) /\ ~(HAS_LOOP G) /\ ~(MULTI_EDGE G)");

let lemma1 = TAC_PROOF([[]],
"((G:Graph) a. 'HAS_LOOP G /\ a IS (EX G) ==> '(a_arc a = a_def a)",
  REPEAT GEM_TAC THEN REWRITE_TAC[HAS_LOOP_DEF;LOOP_DEF]
  THEN CONV_TAC[BUCH_DEPTH_CONV;BUT_EXISTS_CONV]
  THEN STRIP_TAC THEN RES_TAC THEN IMP_RES_TAC[IMP_F]);

let SIMPLE_NO_LOOP = TAC_PROOF([[]],

```

```

"0:"Graph.
SIMPLE_GRAPH d ==> (ta. (a IS (EX G)) ==> ~(a_src a = a_doe a))",
REWRITING_TAC(SIMPLE_GRAPH_DEF)
THEN REPEAT STRIP_TAC THEN IMP_ASK_TAC (ta);];

% - Is to y. (FST x = FST y) /\ (SHD x = SHD y) ==> "(x = y) -%
let pair lemma = GEN_ALL (REWRITING_RULE[SHD, NORMAN_TUP]
(CORTRAPON( Fst(SEQ_IMP_RULE (SPDC.ALL PAIR_EQ)))));];

let eq lemma = TAC_PROOF([[]],
  "(a=b) h (a=c) d. (a = d) /\ (b = c) ==> "(a = b) ==> "(a = c))",
  REPEAT GEN_TAC THEN STRIP_TAC THEN ASSM_REWRITING_TAC[]];

let EDGE_EQ = prove.thm('EDGE_EQ',
  "(a1 `Edge` a2. (a1 = a2) =
  ((a_src a1 = a_src a2) /\ (a_doe a1 = a_doe a2) /\ (a1b a1 = a1b a2))",
  REWRITING_TAC[a_src_DEF, a_doe_DEF, PAIR_EQ, EQ]);];

% Anti-parallel edges in a simple graph are distinct. so
% simple graph is directed -%
let GRAPH_DIRECTED = prove.thm('GRAPH_DIRECTED',
  "(d:"Graph). (SIMPLE_GRAPH d) ==>
  ((a1 a2. (a1 IS (EX G)) /\ (a2 IS (EX G)) /\
  (a_src a1 = a_doe a2) /\ (a_doe a1 = a_src a2) ==> "(a1 = a2))",
  EDGE_REWRITING_TAC(EDGE_EQ) THEN ASS_TAC THEN STRIP_TAC
  THEN REPEAT GEN_TAC THEN STRIP_TAC
  THEN IMP_ASK_TAC SIMPLE_NO_LOOP THEN ASS_REWRITING_TAC[]];];

%-----%
% - A graph is finite iff both vertex set and edge set are finite -%
%-----%
let FINITE_GRAPH_DEF = new_definition('FINITE_GRAPH_DEF',
  "FINITE_GRAPH (d:"Graph) = (GRAPH G) /\
  FINITE (VS G) /\ FINITE (ES G)");];

%-----%
% - Adjacency relations -%
%-----%
% - vertices are adjacent if there is an edge connecting them -%
let VER_ADJA_DEF = new_definition('VER_ADJA_DEF',
  "VER_ADJA G oi (v1 v2) =
  (GRAPH G) /\ (a1 IS_VERTEX G) /\ (a2 IS_VERTEX G) /\
  ((a:"Edge). (a IS_EDGE G) /\ ((a_src a = v1) /\ (a_doe a = v2)) /\
  ((a_src a = v2) /\ (a_doe a = v1))))");];

% - the edges are adjacent if they incident with a common vertex -%
let E_ADJA_DEF = new_definition('E_ADJA_DEF',
  "E_ADJA G oi (a1:"Edge) =
  (GRAPH G) /\ (a1 IS_EDGE G) /\ (a2 IS_EDGE G) /\
  ((a_doe a1 = a_src a2) /\ (a_doe a2 = a_src a1))");];

%-----%
% - Incident relations of vertex -%
%-----%
% - A subset of edges of graph G which is incident from v -%
let INCIDENT_FROM_DEF = new_definition('INCIDENT_FROM_DEF',
  "INCIDENT_FROM (d:"Graph) v =

```

```

(a | (a IS_EDGE G) /\ (a_src a = v))>);;

% the set degree of v is the cardinal of the set INCIDENT_FROM -%
let OUT_DEGREE_DEF = new_definition('OUT_DEGREE_DEF',
  "OUT_DEGREE (G:"Graph) v = CARD (INCIDENT_FROM G v)");;

% Similar for incident to a vertex -%
let INCIDENT_TO_DEF = new_definition('INCIDENT_TO_DEF',
  "INCIDENT_TO (G:"Graph) v =
    (a | (a IS_EDGE G) /\ (a_dest a = v))");;

% the in degree of v is the cardinal of the set INCIDENT_TO -%
let IN_DEGREE_DEF = new_definition('IN_DEGREE_DEF',
  "IN_DEGREE (G:"Graph) v = CARD (INCIDENT_TO G v)");;

% A subset of edges of graph G which is incident with w -%
let INCIDENT_WITH_DEF = new_definition('INCIDENT_WITH_DEF',
  "INCIDENT_WITH (G:"Graph) w =
    (a | (a IS_EDGE G) /\ ((a_src a = w) \/ (a_dest a = w)))");;

% The total degree of a vertex is the sum of the above two -%
let DEGREE_DEF = new_definition('DEGREE_DEF',
  "DEGREE (G:"Graph) v = (IN_DEGREE G v) + (OUT_DEGREE G v)");;

%-----%
% Successor and predecessor relations of vertex -%
%-----%
let IS_SUC_VER G v1 v2 iff v2 is a successor of v1 %
let IS_SUC_VER_DEF = new_definition('IS_SUC_VER_DEF',
  "IS_SUC_VER (G:"Graph) v1 v2 =
    To. (a IS_EDGE G) /\ (a_src a = v1) /\ (a_dest a = v2)");;

let IS_PRE_VER G v1 v2 iff v2 is a predecessor of v1 %
let IS_PRE_VER_DEF = new_definition('IS_PRE_VER_DEF',
  "IS_PRE_VER (G:"Graph) v1 v2 =
    To. (a IS_EDGE G) /\ (a_dest a = v1) /\ (a_src a = v2)");;

% SUC_VERS G v delivers a set of all vertices which are successors of v %
let SUC_VERS_DEF = new_definition('SUC_VERS_DEF',
  "SUC_VERS (G:"Graph) v =
    {v' | (v' IS_VERTEX G) /\ (IS_SUC_VER G v v')}");;

% PRE_VERS G v delivers a set of all vertices which are predecessors of v %
let PRE_VERS_DEF = new_definition('PRE_VERS_DEF',
  "PRE_VERS (G:"Graph) v =
    {v' | (v' IS_VERTEX G) /\ (IS_PRE_VER G v v')}");;

% EDGES_BETWEEN G v1 v2 delivers a set of edges(a)
% all of which are from v1 to v2 %
let EDGES_BETWEEN_DEF = new_definition('EDGES_BETWEEN_DEF',
  "EDGES_BETWEEN (G:"Graph) v1 v2 =
    (a | (a IS_EDGE G) /\ (a_src a = v1) /\ (a_dest a = v2))");;

let g_1mmml = TAC_PROOF([
  "(G:"Graph) (GRAPH G) ==>
    (a. (a IS_EDGE G) ==> ((a_src a) IS (VER G)) /\

```

```

((a_dec a) IS (VS G))",
  DISCH_REWRITING_TAC[GRAPH_EDGEDEF]
  THEN REWRITING_TAC[GRAPH_DEF, VERTEXES_EDGEDEF];];

let VER_INCIDENT_EDGESET = prove.thm('VER_INCIDENT_EDGESET',
  "(G:Graph) v
  (GRAPH G) /\ ~(INCIDENT_WITH G v) = EMPTY" ==> (v IS VERTEX G)",
  REWRITING_TAC[INCIDENT_WITH_DEF, IS_VERTEX_DEF, IS_EDGE_DEF]
  THEN REWRITING_TAC[IS_ALL, (SYM (SPEC_ALL MEMBER_EDGESET))]]
  THEN CONV_TAC (CONV1.CONV SET_SPEC_CONV)
  THEN CONV_TAC (DISCH_CONV1.CONV SYM_CONV)
  THEN REPEAT GEN_TAC THEN STRIP_TAC
  THEN IMP_RES_TAC g.1.lemma1 THEN ASS_REWRITING_TAC[]];];

let NOT_VER_INCIDENT_EDGESET = prove.thm('NOT_VER_INCIDENT_EDGESET',
  "(G:Graph) v
  (GRAPH G) ==> ~(v IS VERTEX G) ==> ((INCIDENT_WITH G v) = EMPTY)",
  REPEAT GEN_TAC THEN STRIP_TAC THEN CONV_TAC CONTRAPOSCONV
  THEN REWRITING_TAC[] THEN DISCH_TAC THEN IMP_RES_TAC VER_INCIDENT_EDGESET];];

let GRAPH_EDGE_VERTEX = prove.thm('GRAPH_EDGE_VERTEX',
  "(G:Graph) e. (GRAPH G) /\ (e IS_EDGE G) ==>
  ((e_arc e) IS_VERTEX G) /\ ((e_dec e) IS_VERTEX G)",
  REWRITING_TAC[IS_VERTEX_DEF, IS_EDGE_DEF] THEN REPEAT GEN_TAC THEN STRIP_TAC
  THEN IMP_RES_TAC g.1.lemma1 THEN ASS_REWRITING_TAC[]];];

let NOT_IS_SAME_SET = prove.thm('NOT_IS_SAME_SET',
  "!(a:e) y a. y IS a /\ 'a IS a ==> ~(a=y)",
  REPEAT GEN_TAC THEN ASS_CASES_TAC "a:a = y"
  THEN ASS_REWRITING_TAC[NOT_AND];];

let NOT_IS_SAME_GRAPH = prove.thm('NOT_IS_SAME_GRAPH',
  "(G:Graph) u e.
  (GRAPH G) /\ ~(e IS_VERTEX G) /\ (e IS_EDGE G) ==>
  '(e_arc e = e) /\ ~(e_dec e = e)",
  REWRITING_TAC[IS_VERTEX_DEF, IS_EDGE_DEF]
  THEN REPEAT GEN_TAC THEN STRIP_TAC
  THEN IMP_RES_TAC (REWRITING_EDGE[IS_VERTEX_DEF, IS_EDGE_DEF]
    GRAPH_EDGE_VERTEX)
  THEN IMP_RES_TAC NOT_IS_SAME_SET
  THEN CONV_TAC (DISCH_CONV1.CONV SYM_CONV) THEN ASS_REWRITING_TAC[]];];

let VERTEX_EDGE = prove.thm('VERTEX_EDGE',
  "(G:Graph) v e.
  (GRAPH G) /\ (v IS_VERTEX G) /\ (e IS (INCIDENT_WITH G v)) ==>
  ((e_arc e = e) /\ (e_dec e = v)",
  REWRITING_TAC[INCIDENT_WITH_DEF]
  THEN CONV_TAC (CONV1.CONV SET_SPEC_CONV)
  THEN REPEAT GEN_TAC THEN STRIP_TAC THEN ASS_REWRITING_TAC[]];];

[-----]
[ DELETION --- DELETE_EDGE deletes an edge from the graph -E ]
[ DELETE_VERTEX deletes a vertex and all edges incident with it -V ]
[-----]

let DELETE_EDGE_DEF = use_info_definition('DELETE_EDGE_DEF',
  "DELETE_EDGE (G:Graph) e = ((VS G), ((EN G) DELETE e))";];

```

```

((a_doe a) is (vs d)) = \
  ONCE_REMOVE_TAC[GRAPH_DECOMP]
  THEN REMOVE_TAC[GRAPH_DEF;VERTICES;EDGES];;

let VER_INCIDENT_EMPTY = prove_thm('VER_INCIDENT_EMPTY',
  "((e:Graph) v.
  (GRAPH e) /\ ~(INCIDENT_WITH d v) = EMPTY) ==> (v is VERTEX e)",
  REMOVE_TAC[INCIDENT_WITH_DEF;IS_VERTEX_DEF;IS_EDGE_DEF]
  THEN REMOVE_TAC[GEN.ALL (SYM SPEC ALL MEMBER_EMPTY)];;
  THEN CONV_TAC (EMPTY_CONV SET_SPEC_CONV)
  THEN CONV_TAC (ONCE_REMOVE_CONV STR_CONV)
  THEN REPEAT GEN_TAC THEN STRIP_TAC
  THEN IMP_ABS_TAC g.lemmal THEN ASS_REMOVE_TAC[]];;

let SET_VER_INCIDENT_EMPTY = prove_thm('SET_VER_INCIDENT_EMPTY',
  "((e:Graph) v.
  (GRAPH e) ==> ~(v is VERTEX e) ==> ((INCIDENT_WITH d v) = EMPTY)",
  REPEAT GEN_TAC THEN STRIP_TAC THEN CONV_TAC CONTRAPOSC_CONV
  THEN REMOVE_TAC[] THEN DISCH_TAC THEN IMP_ABS_TAC VER_INCIDENT_EMPTY);;

let GRAPH_EDGE_VERTEX = prove_thm('GRAPH_EDGE_VERTEX',
  "((e:Graph) e. (GRAPH e) /\ (e is EDGE e) ==>
  (e_arc e) is VERTEX e) /\ ((a_doe a) is VERTEX e)",
  REMOVE_TAC[IS_VERTEX_DEF;IS_EDGE_DEF] THEN REPEAT GEN_TAC THEN STRIP_TAC
  THEN IMP_ABS_TAC g.lemmal THEN ASS_REMOVE_TAC[]];;

let SET_IS_SAME_SET = prove_thm('SET_IS_SAME_SET',
  "(x=a) y a. y is a /\ ~ x is a ==> ~(a=y)",
  REPEAT GEN_TAC THEN ASS_CASES_TAC "a = y"
  THEN ASS_REMOVE_TAC[SET_ABS];;

let SET_IS_SAME_GRAPH = prove_thm('SET_IS_SAME_GRAPH',
  "((e:Graph) v e.
  (GRAPH e) /\ (v is VERTEX e) /\ (e is EDGE e) ==>
  ~(e_arc e = v) /\ ~(a_doe a = v)",
  REMOVE_TAC[IS_VERTEX_DEF;IS_EDGE_DEF]
  THEN REPEAT GEN_TAC THEN STRIP_TAC
  THEN IMP_ABS_TAC (REMOVE_RULE[IS_VERTEX_DEF;IS_EDGE_DEF])
  GRAPH_EDGE_VERTEX
  THEN IMP_ABS_TAC SET_IS_SAME_SET
  THEN CONV_TAC (ONCE_REMOVE_CONV STR_CONV) THEN ASS_REMOVE_TAC[]];;

let VERTEX_EDGE = prove_thm('VERTEX_EDGE',
  "((e:Graph) v e.
  (GRAPH e) /\ (v is VERTEX e) /\ (e is (INCIDENT_WITH d v)) ==>
  ((a_arc a = v) /\ (a_doe a = v)",
  REMOVE_TAC[INCIDENT_WITH_DEF]
  THEN CONV_TAC (EMPTY_CONV SET_SPEC_CONV)
  THEN REPEAT GEN_TAC THEN STRIP_TAC THEN ASS_REMOVE_TAC[]];;

%-----
% DELETION --- DELETE_EDGE deletes an edge from the graph -X
%   DELETE_VERTEX deletes a vertex and all edges incident with it -X
%-----
let DELETE_EDGE_DEF = new_infix_definition('DELETE_EDGE_DEF'
  "DELETE_EDGE (e:(Graph)) d = ((vs d), ((re d) DELETE a))";;

```



```

let DELETE_VERTEX_DEF = new_infra_definition('DELETE_VERTEX_DEF',
  "DELETE_VERTEX (d:Graph) v =
  (((v < d) DELETE v), ((d < d) DIFF (INCIDENT_WITH d v))))";

let E_DELETE_AROUND = prove_thm('E_DELETE_AROUND',
  "!(G:Graph) e.
  (GRAPH G) /\ ~(e IS_EDGE G) ==> ((DELETE_EDGE G e) = G)",
  REWRITING_TAC[IS_EDGE_DEF;DELETE_EDGE_DEF]
  THEN REPEAT GEN_TAC THEN STRIP_TAC
  THEN IMP_RES_TAC DELETE_IS_EDGE THEN ASS_REWRITING_TAC[]
  THEN IMP_RES_THEN (\lam. MATCH_ACCEPT_TAC (BYE lam)) GRAPH_PAIRS[]);

let V_DELETE_AROUND = prove_thm('V_DELETE_AROUND',
  "!(G:Graph) v.
  (GRAPH G) /\ ~(v IS_VERTEX G) ==> ((DELETE_VERTEX G v) = G)",
  REWRITING_TAC[IS_VERTEX_DEF;DELETE_VERTEX_DEF]
  THEN REPEAT GEN_TAC THEN STRIP_TAC
  THEN IMP_RES_TAC DELETE_IS_VERTEX
  THEN IMP_RES_TAC (REWRITING_RULE[IS_VERTEX_DEF] NOT_VER_INCIDENT_DEFPT)
  THEN ASS_REWRITING_TAC[DIFF_DEFPT]
  THEN IMP_RES_THEN (\lam. MATCH_ACCEPT_TAC (BYE lam)) GRAPH_PAIRS[]);

let GRAPH_DELETE_EDGE = prove_thm('GRAPH_DELETE_EDGE',
  "!(G:Graph) (e:Edge).
  (GRAPH G) ==> (GRAPH (G DELETE_EDGE e))",
  REWRITING_TAC[DELETE_EDGE_DEF]
  THEN REWRITING_TAC[GRAPH_DELETE]
  THEN REPEAT STRIP_TAC THEN RES_TAC THEN ASS_REWRITING_TAC[]);

let GRAPH_DELETE_VERTEX = prove_thm('GRAPH_DELETE_VERTEX',
  "!(G:Graph) (v:Vertex).
  (GRAPH G) ==> (GRAPH (G DELETE_VERTEX v))",
  PUSH_DOWN_REWRITING_TAC[DELETE_VERTEX_DEF]
  THEN PUSH_DOWN_REWRITING_TAC[GRAPH_DELETE]
  THEN REWRITING_TAC[VERTICES_DELETE]
  THEN GRAPH_DEF;IS_DELETE;IS_DIFF;INCIDENT_WITH_DEF]
  THEN CHVY_TAC (DEPTVS_CHVY SET_SPDC_CHVY)
  THEN REWRITING_TAC[IS_EDGE_DEF;DE_NORMAL_THM]
  THEN REPEAT GEN_TAC THEN STRIP_TAC THEN GEN_TAC THEN STRIP_TAC
  THEN RES_TAC THEN ASS_REWRITING_TAC[]);

let DELETE_VERTEX_CONN = prove_thm('DELETE_VERTEX_CONN',
  "!(G:Graph) v1 v2.
  ((G DELETE_VERTEX v1) DELETE_VERTEX v2) =
  ((G DELETE_VERTEX v2) DELETE_VERTEX v1)",
  REWRITING_TAC[DELETE_VERTEX_DEF;VERTICES_DELETE;PAIR_DEF]
  THEN REPEAT GEN_TAC THEN CHVY_TAC THEML[
    MATCH_ACCEPT_TAC DELETE_CONN;
    REWRITING_TAC[DIFF_DEF;INCIDENT_WITH_DEF;IS_EDGE_DEF;EDMS]
    THEN CHVY_TAC (DEPTVS_CHVY SET_SPDC_CHVY)
    THEN REWRITING_TAC[EXTENSION] THEN CHVY_TAC (DEPTVS_CHVY SET_SPDC_CHVY)
    THEN REWRITING_TAC[DE_NORMAL_THM] THEN GEN_TAC THEN EQ_TAC
    THEN STRIP_TAC THEN RES_TAC THEN ASS_REWRITING_TAC[]];);

let DELETE_EDGE_CONN = prove_thm('DELETE_EDGE_CONN',
  "!(G:Graph) e1 e2

```

```

    ((G DELETE_EDGE e1) DELETE_EDGE e2) =
      ((G DELETE_EDGE e2) DELETE_EDGE e1)~.
    REMOVE_YAC[DELETE_EDGE_HNF_VERTICES;EDGES;PAIR_BQ]
    THEN MATCH_ACCEPT_YAC DELETE_CONN;;

%-----
% INSERTION --- adding vertex or edge to a graph %
%-----
% INSERT_VERTEX adds a vertex into a graph %
let INSERT_VERTEX_DEF = new.infix.definition('INSERT_VERTEX_DEF',
  "INSERT_VERTEX (G :Graph) = (v INSERT (VS G), (ES G))";);

let GRAPH_INSERT_VERTEX = prove.thm('GRAPH_INSERT_VERTEX',
  "!(G :Graph) v. (GRAPH G) => GRAPH(v INSERT_VERTEX G)~.
  REMOVE_YAC[INSERT_VERTEX_DEF] THEN ONCE REMOVE_YAC[GRAPH_REMOVE]
  THEN REMOVE_YAC[GRAPH_DEF;ES_DEF;VS_DEF;IS_INSERT]
  THEN REPEAT STRIP_TAC THEN RES_TAC THEN ASS REMOVE_YAC[]););

let INSERT_EDGE_DEF = new.infix.definition('INSERT_EDGE_DEF',
  "INSERT_EDGE (G :Graph) = ((VS G),
    (( (e,arc) IS_VERTEX G) /\ ((e,dee) IS_VERTEX G) =>
      (e INSERT (ES G)) /\ (ES G))";);

let GRAPH_INSERT_EDGE = prove.thm('GRAPH_INSERT_EDGE',
  "!(G :Graph) e. (GRAPH G) => GRAPH(e INSERT_EDGE G)~.
  REMOVE_YAC[INSERT_EDGE_DEF] THEN ONCE REMOVE_YAC[GRAPH_REMOVE]
  THEN REMOVE_YAC[GRAPH_DEF;IS_VERTEX_DEF;VERTICES;EDGES]
  THEN REPEAT RES_TAC THEN STRIP_TAC THEN COND_CASES_TAC THEN[(
    REMOVE_YAC[IS_INSERT] THEN RES_TAC THEN STRIP_TAC
    THEN RES_TAC THEN ASS REMOVE_YAC[];
    FIRST_ASSUM MATCH_ACCEPT_TAC)];);

let INSERT_VERTEX_CONN = prove.thm('INSERT_VERTEX_CONN',
  "!(G :Graph) e1 w2.
  (v1 INSERT_VERTEX (v2 INSERT_VERTEX G)) =
    (v2 INSERT_VERTEX (v1 INSERT_VERTEX G))~.
  REMOVE_YAC[INSERT_VERTEX_DEF;VERTICES;EDGES;IS_VERTEX_DEF;PAIR_BQ]
  THEN REPEAT RES_TAC THEN MATCH_ACCEPT_YAC INSERT_CONN;;

let INSERT_EDGE_CONN = prove.thm('INSERT_EDGE_CONN',
  "!(G :Graph) e1 e2.
  (e1 INSERT_EDGE (e2 INSERT_EDGE G)) = (e2 INSERT_EDGE (e1 INSERT_EDGE G))~.
  REMOVE_YAC[INSERT_EDGE_DEF;VERTICES;EDGES;IS_VERTEX_DEF;PAIR_BQ]
  THEN REPEAT RES_TAC THEN COND_CASES_TAC THEN COND_CASES_TAC
  THEN PUSH_REMOVE_YAC[COND_CLAUSES]
  THEN (MATCH_ACCEPT_YAC INSERT_CONN OR LEM REPEAT_TAC)];);

let EDGES_IS_INSERT_VERTEX = prove.thm('IS_INSERT_VERTEX',
  "!(G :Edge) e. G (e IS_EDGE G) => (e IS_EDGE (v INSERT_VERTEX G))~.
  REMOVE_YAC[IS_REMOVE_HNF;EDGES_INSERT_VERTEX_DEF] THEN REPEAT RES_TAC
  THEN COND_CASES_TAC THEN REMOVE_YAC[IS_INSERT;ON_1ST2_TH2];);

let EDGES_IS_INSERT_EDGE = prove.thm('IS_INSERT_EDGE',
  "!(G :Edge) e' G. (e IS_EDGE G) => (e IS_EDGE (e' INSERT_EDGE G))~.
  REMOVE_YAC[IS_REMOVE_HNF;EDGES_INSERT_EDGE_DEF] THEN REPEAT RES_TAC
  THEN COND_CASES_TAC THEN REMOVE_YAC[IS_INSERT;ON_1ST2_TH2];);

```

```

let INCIDENT_WITH_INSERT_VERTEX = prove_thm('INCIDENT_WITH_INSERT_VERTEX',
  "(G:Graph) v. (GRAPH G) => ~(v IS_VERTEX G) ==>
  ((INCIDENT_WITH (v INSERT_VERTEX G) v) = {})"
  REPEAT ONE_TAC THEN REPEAT STRIP_TAC
  THEN IMP_RES_TAC NOT_VER_INCIDENT_EMPTY
  THEN RESWITE_TAC[INSERT_VERTEX_DEF:INCIDENT_WITH_DEF,
  IS_EDGE_DEF:EDGES:EXTENSION]
  THEN CONV_TAC (DEPTH_CONV SET_SPC_CONV)
  THEN ONE_TAC THEN EQ_TAC THENL[
    STRIP_TAC THEN IMP_RES_TAC (RESWITE_MERGE[IS_EDGE_DEF NOT_IN_SAME_GRAPH]:
    RESWITE_TAC[NOT_IN_EMPTY]);];

let lemma1 =
  let e = (ONCE RESWITE_MERGE[EQ_STR_EQ] IS_EDGE_DEF) in
  (RESWITE_MERGE[EQ] (SPWC "(G G):('Edge)set"
    (SPWC "e:('Edge) (INST_TYPE ['"Edge", "e"]) INSERT_RELATE)))));

let DELETE_INSERT_EDGE = prove_thm('DELETE_INSERT_EDGE',
  "(G:Graph) e. (GRAPH G) /\ (e IS_EDGE G) ==>
  ((e INSERT_EDGE G) DELETE_EDGE G) = G",
  RESWITE_TAC[INSERT_EDGE_DEF:DELETE_EDGE_DEF:VERTICES:EDGES]
  THEN REPEAT ONE_TAC THEN STRIP_TAC
  THEN IMP_RES_THEN MP_TAC GRAPH_EDGE_VERTEX
  THEN RESWITE_TAC[IS_VERTEX_DEF:VERTICES:EDGES]
  THEN REPEAT STRIP_TAC THEN RES_TAC THEN ASS RESWITE_TAC[]
  THEN IMP_RES_THEN (λe. RESWITE_TAC[{}]) lemma1
  THEN CONV_TAC STR_CONV THEN IMP_RES_THEN MATCH_ACCEPT_TAC GRAPH_PAIR);

let INSERT_DELETE_VERTEX = prove_thm('INSERT_DELETE_VERTEX',
  "(G:Graph) v. (GRAPH G) /\ (v IS_VERTEX G) ==>
  (((v INSERT_VERTEX G) DELETE_VERTEX v) = G)",
  REPEAT ONE_TAC THEN STRIP_TAC
  THEN IMP_RES_TAC INCIDENT_WITH_INSERT_VERTEX
  THEN RESWITE_TAC[INSERT_VERTEX_DEF:DELETE_VERTEX_DEF:VERTICES:EDGES]
  THEN IMP_RES_THEN (λv. CONV_TAC (ONCE_DEPTH_CONV (RESWITE_CONV v)))
    GRAPH_PAIR
  THEN RESWITE_TAC[PAIR, EQ:VERTICES:EDGES] THEN ONE_TAC THENL[
    RESWITE_TAC[DELETE_INSERT]
    THEN ASSUM_LIST (λmm1. MP_TAC (RESWITE_MERGE[IS_VERTEX_DEF] (e1 2 mm1)))
    THEN RESWITE_TAC[DELETE_MERGE_ELEMENT];
    POP_ASSUM (λv. RESWITE_TAC[RESWITE_MERGE[INSERT_VERTEX_DEF:VERTICES:EDGES] e]);
    THEN MATCH_ACCEPT_TAC DIFF_EMPTY);];

let VERTICES_INSERT_EDGE = prove_thm('VERTICES_INSERT_EDGE',
  "(G:Graph) e. V(e INSERT_EDGE G) = V(e G",
  REPEAT ONE_TAC THEN RESWITE_TAC[INSERT_EDGE_DEF:VERTICES]);];

let EDGES_INSERT_VERTEX = prove_thm('EDGES_INSERT_VERTEX',
  "(G:Graph) v. E(v INSERT_VERTEX G) = E(G",
  REPEAT ONE_TAC THEN RESWITE_TAC[INSERT_VERTEX_DEF:EDGES]);];

let VERTEX_INSERT_VERTEX = prove_thm('VERTEX_INSERT_VERTEX',
  "(G:Graph) x y.
  (x IS_VERTEX (y INSERT_VERTEX G)) = (x = y) /\ (x IS_VERTEX G)",
  RESWITE_TAC[INSERT_VERTEX_DEF:IS_VERTEX_DEF:VERTICES:IS_INSERT]);];

let EDGE_INSERT_EDGE = prove_thm('EDGE_INSERT_EDGE',

```

```

"((G:"Graph) a.
  ((a_src a) IS_VERTEX G) /\ ((a_dest a) IS_VERTEX G) ==>
  (a IS_EDGE (a INSERT_EDGE G))",
  REPEAT STRIP_TAC THEN
  ASS. ASSUMES_TAC[INSERT_EDGE_DEF; IS_EDGE_DEF; EDGEIN IS_INSERT]);

let EDGE_IN_INSERT = prove_thm('EDGE_IN_INSERT',
  "((G:"Graph) a. ((a_src a) IS_VERTEX G) /\ ((a_dest a) IS_VERTEX G)
  ==> (a IS_EDGE (a INSERT_EDGE G))",
  REPEAT STRIP_TAC THEN PURE. ASS. ASSUMES_TAC[INSERT_EDGE_DEF]
  THEN ASS. ASSUMES_TAC[IS_EDGE_DEF; EDGEIN IS_INSERT]);

let EDGE_IN_INSERT2 = save_thm('EDGE_IN_INSERT2',
  PURE. ASS. ASSUMES_TAC[a_src:a_dest]
  (OBLI ["a:"Graph"; "v1:a"; "v2:a"; "u:a"]
  (ISPEC ("(v1,v2,u):Edge" (ISPEC "G:"Graph" EDGEIN IS_INSERT))));

let VERTEX_IN_IS_VERTEX = prove_thm('VERTEX_IN_IS_VERTEX',
  "((G:"Graph) v u.
  (v IS_VERTEX (u INSERT_VERTEX G)) = (u = v) /\ (v IS_VERTEX G)",
  ASSUMES_TAC[INSERT_VERTEX_DEF; IS_VERTEX_DEF; VERTEXIN IS_INSERT]);

let IS_INSERT_AROUND = TAC_PROOF([[]
  "!(a:o). (a IS a) ==> ((a INSERT a) = a)",
  PURE. ASSUMES_TAC[INSERT_DEF; EXTENSION]
  THEN REPEAT STRIP_TAC THEN CHV. TAC (CHC. DEPTH. CHV. SKY. SPEC. CHV)
  THEN EQ. TAC THEN STRIP_TAC THEN ASS. ASSUMES_TAC[]]);

let V_INSERT_AROUND = prove_thm('V_INSERT_AROUND',
  "((G:"Graph) u. (GRAPH G) /\ (v IS_VERTEX G) ==>
  ((v INSERT_VERTEX G) = G)",
  ASSUMES_TAC[INSERT_VERTEX_DEF; IS_VERTEX_DEF]
  THEN REPEAT STRIP_TAC THEN IMP. RES. THEN SUBST1_TAC GRAPH_PAIR
  THEN PURE. ASSUMES_TAC[VERTECH. EDGEIN]
  THEN IMP. RES. THEN SUBST1_TAC IS_INSERT_AROUND THEN REFL. TAC]);

let E_INSERT_AROUND = prove_thm('E_INSERT_AROUND',
  "((G:"Graph) a. (GRAPH G) /\ (a IS_EDGE G) ==>
  (a INSERT_EDGE G) = G",
  ASSUMES_TAC[INSERT_EDGE_DEF] THEN REPEAT STRIP_TAC
  THEN IMP. RES. TAC GRAPH_EDGE_VERTEX THEN ASS. ASSUMES_TAC[]
  THEN IMP. RES. TAC IS_EDGE_DEF THEN IMP. RES. THEN SUBST1_TAC IS_INSERT_AROUND
  THEN CHV. TAC (CHC. DEPTH. CHV. SKY. CHV) THEN IMP. RES. TAC GRAPH_PAIR);

let FINITE_GRAPH_INSERT_EDGE = prove_thm('FINITE_GRAPH_INSERT_EDGE',
  "((G:"Graph) a. FINITE_GRAPH G ==> FINITE_GRAPH(a INSERT_EDGE G)",
  PURE. ASS. ASSUMES_TAC[FINITE_GRAPH_DEF]
  THEN REPEAT STRIP_TAC THEN [
    IMP. RES. THEN (V. MATCH. ACCEPT. TAC a) GRAPH_INSERT_EDGE;
    PURE. ASSUMES_TAC[VERTECH. INSERT_EDGE_DEF]
    THEN FINITE_GRAPH_ACCEPT. TAC;
    PURE. ASSUMES_TAC[EDGEIN IS_INSERT_EDGE_DEF]
    THEN CHV. CASEN. TAC THEN PURE. ASS. ASSUMES_TAC[FINITE_INSERT]);];

```

```

%-----%
% 4. INTER --- interaction of two graphs %
%-----%

```

```

let G_INTER_DEF = new_infra_definition('G_INTER_DEF',
  "G_INTER (G1:"Graph) G2 =
    (((VS G1) INTER (VS G2)), ((ES G1) INTER (ES G2)))");;

let GRAPH_INTER = prove_thm('GRAPH_INTER',
  "(G1:"Graph) G2.
  (GRAPH G1) /\ (GRAPH G2) ==> (GRAPH (G1 G_INTER G2))",
  PURE_ONCE_REWRITE_TAC[GRAPH_UNCHOP]
  THEN PURE_ONCE_REWRITE_TAC[G_INTER_DEF]
  THEN PURE_ONCE_REWRITE_TAC[VERTICES_EDGEK]
  THEN PURE_ONCE_REWRITE_TAC[GRAPH_DEF]
  THEN REPEAT GEN_TAC THEN STRIP_TAC THEN REWRITE_TAC[IN_INTER]
  THEN GEN_TAC THEN STRIP_TAC THEN RES_TAC THEN ASS_REWRITE_TAC[]);;

let G_INTER_IDENT = prove_thm('G_INTER_IDENT',
  "(G:"Graph) (G G_INTER G) = G",
  GEN_TAC THEN REWRITE_TAC[G_INTER_DEF][INTER_IDENTFC][VS_DEF][ES_DEF]);;

let G_INTER_SYM = prove_thm('G_INTER_SYM',
  "(G1:"Graph) G2. (G1 G_INTER G2) = (G2 G_INTER G1)",
  REPEAT GEN_TAC THEN REWRITE_TAC[G_INTER_DEF][PAIR_EQ][VERTICES_EDGEK]
  THEN CUR1_TAC THEN MATCH_ACCEPT_TAC[INTER_COMM]);;

let G_INTER_ASSOC = prove_thm('G_INTER_ASSOC',
  "(G1:"Graph) G2 G3.
  ((G1 G_INTER G2) G_INTER G3) = (G1 G_INTER (G2 G_INTER G3))",
  REPEAT GEN_TAC THEN REWRITE_TAC[G_INTER_DEF][PAIR_EQ][VERTICES_EDGEK]
  THEN CUR1_TAC THEN MATCH_ACCEPT_TAC[INTER_ASSOC]);;

let VERTX_IS_INTER = prove_thm('VERTX_IS_INTER',
  "(G1:"Graph) G2 v.
  (v IS_VERTX (G1 G_INTER G2)) = ((v IS_VERTX G1) /\ (v IS_VERTX G2))",
  REPEAT GEN_TAC
  THEN REWRITE_TAC[G_INTER_DEF][IS_VERTX_DEF][SERVICES_IS_INTER]);;

let EDGE_IS_INTER = prove_thm('EDGE_IS_INTER',
  "(G1:"Graph) G2 e.
  (e IS_EDGE (G1 G_INTER G2)) = ((e IS_EDGE G1) /\ (e IS_EDGE G2))",
  REPEAT GEN_TAC
  THEN REWRITE_TAC[G_INTER_DEF][IS_EDGE_DEF][EDGEK_IS_INTER]);;

[-----]
% G_UNION --- union of two graphs %
[-----]

let G_UNION_DEF = new_infra_definition('G_UNION_DEF',
  "G_UNION (G1:"Graph) G2 =
    (((VS G1) UNION (VS G2)), ((ES G1) UNION (ES G2)))");;

let GRAPH_UNION = prove_thm('GRAPH_UNION',
  "(G1:"Graph) G2.
  (GRAPH G1) /\ (GRAPH G2) ==> (GRAPH (G1 G_UNION G2))",
  ONCE_REWRITE_TAC[GRAPH_UNCHOP] THEN ONCE_REWRITE_TAC[G_UNION_DEF]
  THEN ONCE_REWRITE_TAC[VERTICES_EDGEK] THEN ONCE_REWRITE_TAC[GRAPH_DEF]
  THEN REPEAT GEN_TAC THEN STRIP_TAC THEN REWRITE_TAC[IS_UNION]
  THEN GEN_TAC THEN STRIP_TAC THEN RES_TAC THEN ASS_REWRITE_TAC[]);;

let G_UNION_IDENT = prove_thm('G_UNION_IDENT',

```

```

"!(0:"Graph). (Q Q_UNION Q) = Q",
GEN_TAC THEN REWRITE_TAC[Q_UNION_DEF;UNION_INPUT.VK_DEF;IN_DEF];;

let Q_UNION_SYM = prove_thm('Q_UNION_SYM',
  "!(01:"Graph) Q2. (Q1 Q_UNION Q2) = (Q2 Q_UNION Q1)",
  REPEAT GEN_TAC THEN REWRITE_TAC[Q_UNION_DEF;PAIR_EQ]
  THEN CORJ_TAC THEN MATCH_ACCEPT_TAC UNION_COMM);;

let Q_UNION_ASSOC = prove_thm('Q_UNION_ASSOC',
  "!(01:"Graph) Q2 Q3.
  ((Q1 Q_UNION Q2) Q_UNION Q3) = (Q1 Q_UNION (Q2 Q_UNION Q3))",
  REPEAT GEN_TAC THEN REWRITE_TAC[Q_UNION_DEF;PAIR_EQ;VERTICES_EDGEIN]
  THEN CORJ_TAC THEN MATCH_ACCEPT_TAC UNION_ASSOC);;

let VERTICES_IN_UNION = prove_thm('VERTICES_IN_UNION',
  "!(01:"Graph) Q2 v1 v2.
  (GRAPH Q1) /\ (GRAPH Q2) /\
  (v1 IS_VERTEX Q1) /\ (v2 IS_VERTEX Q2) ==>
  ((v1 IS_VERTEX (Q1 Q_UNION Q2)) /\ (v2 IS_VERTEX (Q1 Q_UNION Q2)))",
  REPEAT GEN_TAC THEN REWRITE_TAC[Q_UNION_DEF;IS_VERTEX_DEF;VERTICES_EDGEIN]
  THEN STRIP_TAC THEN ASS_REWRITE_TAC[IS_UNION]);;

let VERTEX_IN_UNION = prove_thm('VERTEX_IN_UNION',
  "!(01:"Graph) Q2 v.
  ((v IS_VERTEX (Q1 Q_UNION Q2)) = ((v IS_VERTEX Q1) /\ (v IS_VERTEX Q2)))",
  REPEAT GEN_TAC
  THEN REWRITE_TAC[Q_UNION_DEF;IS_VERTEX_DEF;VERTICES_IN_UNION]);;

let EDGE_IN_UNION = prove_thm('EDGE_IN_UNION',
  "!(01:"Graph) Q2 e.
  ((e IS_EDGE (Q1 Q_UNION Q2)) = ((e IS_EDGE Q1) /\ (e IS_EDGE Q2)))",
  REPEAT GEN_TAC THEN REWRITE_TAC[Q_UNION_DEF;IS_EDGE_DEF;EDGES_IN_UNION]);;

let VERTEX_INSERT_EDGE = prove_thm('VERTEX_INSERT_EDGE',
  "!(0:"Graph) v n. (v IS_VERTEX (INSERT_EDGE Q)) = (v IS_VERTEX Q)",
  REPEAT GEN_TAC THEN REWRITE_TAC[INSERT_EDGE_DEF;IS_VERTEX_DEF;VERTICES]);;

% lemma1 = |- !Q v1 v2.
  GRAPH Q /\ v1 IN (VS Q) /\ v2 IN (VS Q) ==>
  GRAPH ((v1,v2,n) INSERT_EDGE Q) %
let lemma1 = REWRITE_RULE[a_esc_DEF;a_def_DEF;PST_DEF] (GEN_ALL
  (SPEC "(v1,v2,n1) :>Edge" (SPEC Q:"Graph" GRAPH_INSERT_EDGE)));;

let lemma2 = REWRITE_RULE[a_esc_DEF;a_def_DEF;PST_DEF] (GEN_ALL
  (SPEC "(v2,v1,n2) :>Edge" (SPEC Q:"Graph" GRAPH_INSERT_EDGE)));;

let GRAPH_INSERT_EDGES = prove_thm('GRAPH_INSERT_EDGES',
  "!(0:"Graph) v1 v2.
  (GRAPH Q) /\ (v1 IS_VERTEX Q) /\ (v2 IS_VERTEX Q) ==>
  (!a1. GRAPH((v1,v2,a1) INSERT_EDGE Q)) /\
  (!a2. GRAPH((v2,v1,a2) INSERT_EDGE Q))",
  REPEAT GEN_TAC THEN STRIP_TAC THEN CORJ_TAC THEN GEN_TAC
  THEN INP_BES_TAC lemma1 THEN INP_BES_TAC lemma2 THEN ASS_REWRITE_TAC[]);;

let lemma1 = GEN_ALL (SPEC "(Q1 Q_UNION Q2) :>Graph" GRAPH_INSERT_EDGES);;

let Q_UNION_INSERT_EDGES = prove_thm('Q_UNION_INSERT_EDGES',

```

```

"(G1:"Graph) G2 v1 v2
(GRAPH G1) /\ (GRAPH G2) /\
(v1 IS_VERTEX G1) /\ (v2 IS_VERTEX G2) ==>
{!(a1: GRAPH ((v1,v2,a1) INSERT_EDGE (G1 G_UNION G2)))) /\
{!(a2: GRAPH ((v2,v1,a2) INSERT_EDGE (G1 G_UNION G2))))".
REPEAT GEN_TAC THEN STRIP_TAC THEN MATCH_MP_TAC lemma1
THEN CONJ_TAC THENL [
  IMP_RES_TAC GRAPH_UNION;
  MATCH_MP_TAC VERTEXES_IN_UNION THEN ASS_REWRITE_TAC[]];;

let G_INSERT_E = prove(thm("G_INSERT_E",
  "(G:"Graph) a1 a2.
  (GRAPH (a1 INSERT_EDGE G)) /\ (GRAPH (a2 INSERT_EDGE G)) ==>
  (GRAPH (a1 INSERT_EDGE (a2 INSERT_EDGE G))) /\
  (GRAPH (a2 INSERT_EDGE (a1 INSERT_EDGE G))))".
  REPEAT STRIP_TAC THEN IMP_RES_TAC GRAPH_INSERT_EDGE
  THEN FIRST_ASSUM MATCH_ACCEPT_TAC];;

let GRAPH_INSERT_EDGE2 = TAC_PROOF([],
  "(G:"Graph) v1 v2 a1 a2.
  (GRAPH G) /\ (v1 IS_VERTEX G) /\ (v2 IS_VERTEX G) ==>
  (GRAPH((v1,v2,a1) INSERT_EDGE G)) /\
  (GRAPH((v2,v1,a2) INSERT_EDGE G)))".
  REPEAT GEN_TAC THEN STRIP_TAC THEN IMP_RES_TAC lemma1
  THEN IMP_RES_TAC lemma2 THEN ASS_REWRITE_TAC[]];;

let lemma1 = GEN_ALL (SPEC "(G1 G_UNION G2) : 'Graph' GRAPH_INSERT_EDGE2);;

let G_UNION_INSERT_EDGE2 = TAC_PROOF([],
  "(G1:"Graph) G2 v1 v2 a1 a2.
  (GRAPH G1) /\ (GRAPH G2) /\
  (v1 IS_VERTEX G1) /\ (v2 IS_VERTEX G2) ==>
  (GRAPH((v1,v2,a1) INSERT_EDGE (G1 G_UNION G2))) /\
  (GRAPH((v2,v1,a2) INSERT_EDGE (G1 G_UNION G2))))".
  REPEAT GEN_TAC THEN STRIP_TAC THEN MATCH_MP_TAC lemma1
  THEN CONJ_TAC THENL [
    IMP_RES_TAC GRAPH_UNION;
    MATCH_MP_TAC VERTEXES_IN_UNION THEN ASS_REWRITE_TAC[]];;

% G_UNION_INSERT_EDGE = |-
  "(G1:"Graph) G2 v1 v2. (GRAPH G1) /\ (GRAPH G2) /\
  (v1 IS_VERTEX G1) /\ (v2 IS_VERTEX G2) ==>
  {!(a1 a2: GRAPH ((v1,v2,a1) INSERT_EDGE ((v2,v1,a2) INSERT_EDGE (G1 G_UNION G2))))}"
let G_UNION_INSERT_EDGE = new(thm("G_UNION_INSERT_EDGE",
  GEN_ALL (DISCH_ALL (GEN "a2==>(GEN "a1==>
    (UNDISCH (IMP_TRANS (SPEC_ALL G_UNION_INSERT_EDGE2)
      (SPEC "(v2,v1,a2) : 'Edge'" (SPEC "(v1,v2,a1) : 'Edge'"
        (SPEC "(G1:"Graph) G_UNION G2" G_INSERT_E))))))));;

```

B.3 The file mk_subgraph.ml

```

%-----%
% FILE: mk_subgraph.ml %
% DESCRIPTION: definition of subgraph and some theorems %

```

```

% AUTHOR: Nai Wang DATE: 1 AUG 1990 modified Jan 91 %
%-----%
% Definition of a subgraph %
% H is a subgraph of G iff H is a graph and its vertex set is a subset %
% of that of G and its edge set is also a subset of the edge set of G. %
%-----%
let SUBGRAPH_DEF = new_definition('SUBGRAPH_DEF',
  "(G:Graph) (H:Graph) =
    ((VH H) SUBSET (VH G)) /\ ((EH H) SUBSET (EH G))");

let SUBGRAPH_REFL = prove_thm('SUBGRAPH_REFL',
  "%G:Graph. GRAPHS G ==> SUBGRAPH G G",
  REWRITE_TAC([SUBGRAPH_DEF; SUBSET_REFL]);

let SUBGRAPH_TRANS = prove_thm('SUBGRAPH_TRANS',
  "%G1:Graph. G2 G3.
  (SUBGRAPH G1 G2) /\ (SUBGRAPH G2 G3) ==> (SUBGRAPH G1 G3)",
  REWRITE_TAC([SUBGRAPH_DEF]) THEN REPEAT GEN_TAC THEN STRIP_TAC
  THEN IMP_RES_TAC SUBSET_TRANS THEN ASS. REWRITE_TAC[]);

let SUBGRAPH_ANTISYM = prove_thm('SUBGRAPH_ANTISYM',
  "%G1:Graph. G2.
  (SUBGRAPH G1 G2) /\ (SUBGRAPH G2 G1) ==> (G1 = G2)",
  REWRITE_TAC([SUBGRAPH_DEF]) THEN REPEAT GEN_TAC THEN STRIP_TAC
  THEN IMP_RES_TAC SUBSET_ANTISYM THEN IMP_RES_TAC GRAPHS_EQ);

let SUBGRAPH_GRAPHS = prove_thm('SUBGRAPH_GRAPHS',
  "%G:Graph. H. (SUBGRAPH H G) ==> (GRAPHS G) /\ (GRAPHS H)",
  REWRITE_TAC([SUBGRAPH_DEF]) THEN REPEAT STRIP_TAC
  THEN FIRST_ASSUM ACCEPT_TAC);

%-----%
% Definition of a proper subgraph %
% H is a proper subgraph of G iff H is a graph and its vertex set is a %
% proper subset of that of G and its edge set is also a proper subset %
% of the edge set of G. %
%-----%
let PROPERGRAPH_DEF = new_definition('PROPERGRAPH_DEF',
  "(G:Graph) (H:Graph) =
    ((SUBGRAPH H G) /\ (((VH H) PROPERSET (VH G)) /\ ((EH H) PROPERSET (EH G))))");

let PROPERGRAPH_SUBGRAPH = prove_thm('PROPERGRAPH_SUBGRAPH',
  "%G:Graph. H:Graph. (PROPERGRAPH H G) ==> (SUBGRAPH H G)",
  PURE_ONCE_REWRITE_TAC([PROPERGRAPH_DEF])
  THEN REPEAT STRIP_TAC THEN FIRST_ASSUM ACCEPT_TAC);

let PROPERGRAPH_IRREFL = prove_thm('PROPERGRAPH_IRREFL',
  "%G:Graph. GRAPHS G ==> ~(PROPERGRAPH G G)",
  REWRITE_TAC([PROPERGRAPH_DEF; SUBGRAPH_DEF; IS_PROPERSET]);

let SUBSET_CASES =
  let lem = GEN_REWRITE_RULE([DISJ, SYM] RECLOSED_WIDEN in
  TAC_PROOF([
    "%(a:(o)set). (a SUBSET a) = (a PROPERSET a) /\ (a = a)",
    REPEAT GEN_TAC THEN REWRITE_TAC([PROPERSET_DEF; IS_PROPERSET]);
  ]))

```



```

THEN B4.TAC THEN[
  DISCH_THEN (\s. PURE_EXISTS_TAC[4.08.CLAIMED]);
  PURE_EXISTS_TAC[HYPERHISEL.SURSET_DEF]
  THEN STRIP_TAC THEN GEN_TAC THEN PURE_ASM_EXISTS_TAC[IMP_CLAIMED]]];

let SUB_PSUB_TRANS = TAC_PROOF([],
  "!(e:(c)set) s u. (s PURSET s) /\ (s PURSET u) ==> (s PURSET u)",
  PURE_EXISTS_EXISTS_TAC[SURSET_CASES] THEN REPEAT STRIP_TAC THEN[
    IMP_RES_TAC PURSET_TRANS;
    UDISCH_TAC "s PURSET (t:(c)set)" THEN ASM_EXISTS_TAC[]];

let PSUB_SUB_TRANS = TAC_PROOF([],
  "!(e:(c)set) s u. (s PURSET s) /\ (s SURSET u) ==> (s PURSET u)",
  PURE_EXISTS_EXISTS_TAC[SURSET_CASES] THEN REPEAT STRIP_TAC THEN[
    IMP_RES_TAC PURSET_TRANS;
    UDISCH_TAC "s PURSET (t:(c)set)" THEN ASM_EXISTS_TAC[]];

let PSUBGRAPH_TRANS = prove_thm('PSUBGRAPH_TRANS',
  "!(G:Graph) G2 G3.
  (PSUBGRAPH G1 G2) /\ (PSUBGRAPH G2 G3) ==> (PSUBGRAPH G1 G3)",
  PURE_EXISTS_TAC[PSUBGRAPH_DEF.SUBGRAPH_DEF]
  THEN REPEAT GEN_TAC THEN STRIP_TAC THEN IMP_RES_TAC PURSET_TRANS
  THEN IMP_RES_TAC SURSET_TRANS THEN IMP_RES_TAC PSUB_SUB_TRANS
  THEN ASM_EXISTS_TAC[]];

let lemma1 = TAC_PROOF([],
  "!(G:Graph). (GRAPH G) ==>
  ((e. a IN (EX G) ==> (a_arc a) IN (VE G) /\ (a_dan a) IN (VE G)))",
  RECH_EXISTS_TAC[GRAPHS_RECHDEF]
  THEN EXISTS_TAC[GRAPHS_DEF.VERTICES;HENSE];

let DELETE_PURSET = TAC_PROOF([],
  "!(e:(c)set) s. (s IN s) ==> (s DELETE s) PURSET s",
  PURE_EXISTS_TAC[PURSET_DEF.HYPERHISEL] THEN REPEAT GEN_TAC
  THEN STRIP_TAC THEN CONJ_TAC THEN[
    MATCH_ACCEPT_TAC DELETE_SURSET;
    CONV_TAC TRY.FORALL_CONV THEN EXISTS_TAC "s"
    THEN ASM_EXISTS_TAC[IS_DELETE]]];

let PSUBGRAPH_DELETE_EDGE = prove_thm('PSUBGRAPH_DELETE_EDGE',
  "!(G:Graph) s.
  (GRAPH G /\ (s IS_EDGE G)) ==> (PSUBGRAPH (G DELETE_EDGE s) G)",
  EXISTS_TAC[SUBGRAPH_DEF.PSUBGRAPH_DEF.DELETE_EDGE_DEF;
  VERTICES;HENSE;IS_EDGE_DEF]
  THEN REPEAT GEN_TAC THEN STRIP_TAC THEN IMP_RES_TAC DELETE_PURSET
  THEN ASM_EXISTS_TAC[SURSET_DEF.DELETE_SURSET]
  THEN ASM_CASES_TAC "(EX (G:Graph)) = {}" THEN[
    ASM_EXISTS_TAC[EMPTY_DELETE_GRAPH_DEF;NOT_IS_EMPTY];
    ASM_EXISTS_TAC[DELETE_SURSET_GRAPH_DEF;IS_DELETE]
    THEN GEN_TAC THEN STRIP_TAC THEN IMP_RES_TAC lemma1
    THEN ASM_EXISTS_TAC[]];

let SUBGRAPH_DELETE_EDGE = prove_thm('SUBGRAPH_DELETE_EDGE',
  "!(G:Graph) s.
  (GRAPH G) ==> (SUBGRAPH (G DELETE_EDGE s) G)",
  EXISTS_TAC[SUBGRAPH_DEF.DELETE_EDGE_DEF.VERTICES_EDGE_DEF]
  THEN REPEAT GEN_TAC THEN STRIP_TAC

```

```

THEN ASS_CURR_TAC "(RE (G:"Graph)) = {}" THENL[
  ASS_REMOVE_TAC[EMPTT,SURSET,EMPTY,DELETE;EMSET,REVL;
    GRAPHS,DEF;NOT,IS,EMPTY];
  ASS_REMOVE_TAC[SURSET,REVL,DELETE,SURSET;GRAPHS,DEF;IS,DELETE]
  THEN GEN_TAC THEN STRIP_TAC THEN IMP_RES_TAC 1mmml
  THEN ASS_REMOVE_TAC[]];];

let DIFF_SURSET = YAC_PROOF([[]
  "(G:(G:Gset)) G. (G DIFF G) SURSET G";
  REPEAT GEN_TAC THEN REMOVE_TAC[SURSET,DEF;IS,DIFF;AND1,THE]]];

let SUBGRAPHS_DELETE_VERTEX = prove_thm('SUBGRAPHS_DELETE_VERTEX',
  "(G:"Graph) G.
  (GRAPHS G ==> (SUBGRAPHS (G DELETE_VERTEX G) G)",
  REMOVE_TAC[SUBGRAPHS,DEF;SUBGRAPHS_DELETE_VERTEX,DEF;
    IS_VERTEX,DEF;IS_EDGE,DEF]
  THEN REPEAT STRIP_TAC THENL[
    REMOVE_TAC[GRAPHS,DEF;IS,DIFF;IS,DELETE;INCIDENT_WITH,DEF;IS_EDGE,DEF]
    THEN CONV_TAC[DEPTH_CONV SET.SPAC_CONV]
    THEN REMOVE_TAC[IN,REMOVE,THE;EMSET]
    THEN GEN_TAC THEN STRIP_TAC THENL[
      RES_TAC;
      IMP_RES_TAC 1mmml THEN ASS_REMOVE_TAC[]];
    POP_ASSUM ACCEPT_TAC;
    REVC_ACCEPT_TAC DELETE_SURSET;
    REVC_ACCEPT_TAC DIFF_SURSET];];

let PMSUBGRAPHS_DELETE_VERTEX = prove_thm('PMSUBGRAPHS_DELETE_VERTEX',
  "(G:"Graph) G.
  (GRAPHS G /\ (V IS_VERTEX G) ==> (PMSUBGRAPHS (G DELETE_VERTEX G) G)",
  REMOVE_TAC[SUBGRAPHS,DEF;PMSUBGRAPHS_DELETE_VERTEX,DEF;
    VERVICES;EMSET;IS_VERTEX,DEF;IS_EDGE,DEF]
  THEN REPEAT GEN_TAC THEN STRIP_TAC THEN IMP_RES_TAC DELETE_PMSURSET
  THEN ASS_REMOVE_TAC[SURSET,REVL,DELETE,SURSET;DIFF,SURSET]
  THEN REMOVE_TAC[GRAPHS,DEF;IS,DIFF;IS,DELETE;INCIDENT_WITH,DEF;IS_EDGE,DEF]
  THEN CONV_TAC[DEPTH_CONV SET.SPAC_CONV]
  THEN REMOVE_TAC[IN,REMOVE,THE] THEN GEN_TAC THEN STRIP_TAC THENL[
    RES_TAC;
    IMP_RES_TAC 1mmml THEN ASS_REMOVE_TAC[V,DEF]]];];

%-----%
% 1- HE SUBGRAPHS creates a subgraph from a graph giving two predicates
% 2- which select the vertices and edges from the original graph.
% 3- There are an additional constraint that the end points of the
% 4- edges in the subgraph must be the vertices in that subgraph, thus
% 5- the following two theorems
%-----%
let HE_SUBGRAPHS_DEF = new_definition('HE_SUBGRAPHS_DEF',
  "HE_SUBGRAPHS (G:"Graph) Gv Gs =
  (Gv | G IS_VERTEX G /\ Gv G),
  (Gs | G IS_EDGE G /\ Gs G /\ Gv (G_SRC G) /\ Gv (G_DEST G))");];

let HE_SUBGRAPHS_GRAPH = prove_thm('HE_SUBGRAPHS_GRAPH',
  "(G:"Graph) Gv Gs.
  (GRAPHS G ==> GRAPH (HE_SUBGRAPH G Gv Gs)",
  REMOVE_TAC[HE_SUBGRAPHS,DEF;GRAPHS,DEF;VERVICES;EMSET]
  THEN CONV_TAC[DEPTH_CONV SET.SPAC_CONV]

```

```

THEN PONE_REWRITE_TAC[IS_VERTEX_DEF;IS_EDGE_DEF]
THEN REPEAT GEN_TAC THEN STRIP_TAC THEN GEN_TAC THEN STRIP_TAC
THEN IMP_RES_TAC !normal THEN ASS_REWRITE_TAC[];;

let RE_SUBGRAPHS_SUBGRAPH = prove_thm('RE_SUBGRAPHS_SUBGRAPH',
  "!(G:'Graph') f <->
  (GRAPH G) ==> SUBGRAPH (RE_SUBGRAPH G f) <->
  REWRITE_TAC[SUBGRAPHS_DEF] THEN REPEAT GEN_TAC THEN STRIP_TAC
  THEN IMP_RES_TAC RE_SUBGRAPHS_GRAPH
  THEN PONE_RES_REWRITE_TAC[IS_OBJECTS]
  THEN REWRITE_TAC[RE_SUBGRAPHS_DEF;VERTICES;EDGES;SUBSET_DEF;
  IS_EDGE_DEF;IS_VERTEX_DEF]
  THEN CONV_TAC (DRPTH_CONV SET.SPAC.CONV) THEN CHG1_TAC
  THEN REPEAT STRIP_TAC THEN FIRST_ASSUM MATCH_ACCEPT_TAC);;

-----
% GRAPH_ISO --- Graph isomorphism %
%-----
new_special_symbol '~>>';;
new_special_symbol '~>';;
new_special_symbol '<=>';;

let GRAPH_ISO_DEF = new_definition('GRAPH_ISO_DEF',
  "GRAPH ISO (G:'Graph') (H:'Graph') (f,g) =
  (GRAPH G) /\ (GRAPH H) /\ ((\x y) <-> (f x) /\ ((H y) <-> (H (f y))g));;

let GRAPH_ISO_AUTO = prove_thm('GRAPH_ISO_AUTO',
  "!(G:'Graph') GRAPH G ==> GRAPH ISO G G (I,I)",
  REWRITE_TAC[GRAPH_ISO_DEF;PONE.1]);;

let GRAPH_ISO_TRANS = prove_thm('GRAPH_ISO_TRANS',
  "!(G:'Graph') (G2:'Graph') (G3:'Graph') f1 g1 f2 g2.
  (GRAPH ISO G1 G2 (f1,g1)) /\ (GRAPH ISO G2 G3 (f2,g2)) ==>
  (GRAPH ISO G1 G3 ((G2 o f1), (g2 o g1)))",
  PONE.1UCH.REWRITE_TAC[GRAPH_ISO_DEF] THEN REPEAT STRIP_TAC THEN!
  ALL_TAC; ALL_TAC;
  IMP_RES_TAC PONE.ISO.o; IMP_RES_TAC PONE.ISO.o]
  THEN FIRST_ASSUM (\th g. ACCEPT_TAC th g));;

let GRAPH_ISO_SYM = prove_thm('GRAPH_ISO_SYM',
  "!(G:'Graph') (H:'Graph') f g. (GRAPH ISO G H (f,g)) ==>
  (f' g'. (GRAPH ISO H G (g',f'))",
  PONE.1UCH.REWRITE_TAC[GRAPH_ISO_DEF] THEN REPEAT STRIP_TAC
  THEN EXISTN_TAC "(PONE.1INV (H (G:'Graph')) (H (H:'Graph')) f)"
  THEN EXISTN_TAC "(PONE.1INV (H (G:'Graph')) (H (H:'Graph')) g)"
  THEN IMP_RES_TAC ISO.1INV THEN ASS_REWRITE_TAC[]);;

let GRAPH_ISO_SYM_INV = prove_thm('GRAPH_ISO_SYM_INV',
  "!(G:'Graph') (H:'Graph') f g. (GRAPH ISO G H (f,g)) ==>
  (GRAPH ISO H G ((PONE.1INV (H G)) (H H) f), (PONE.1INV (H G) (H H) g));",
  PONE.1UCH.REWRITE_TAC[GRAPH_ISO_DEF] THEN REPEAT STRIP_TAC
  THEN IMP_RES_TAC ISO.1INV THEN FIRST_ASSUM ACCEPT_TAC);;

close_theory();;

```

B.4 The file mk.elist.ml

```

new_theory'elist':;

load_library'sets';
new_parent'graph';;
setvalued_all'graph';;

set_flag('sticky',true);;

let Vortex = "c" and
    Edge = "(c @ @ cc)" and
    Graph = "(c)set @ (c @ @ cc)set";;

let NULL_EL = prove_thm('NULL_EL',
  "{():}list. NULL 1 = (1 = [])",
  LIST_INDUCT_TAC THEN ASSUME_TAC[NULL:NOV_CONV_EL]);;

let IS_IMP_IS_UNION = TAC_PROOF([],
  "(a@b) @ c. (a IS a) ==> a IS (a UNION b)",
  ASSUME_TAC[IS:IS_IMP_IS_UNION2]);;

%-----%
% Membership of list--- in analogy with set membership (ID) %
% ELIN 1 a is TRUE iff a is an element of the list 1 %
%-----%
let ELIN_DEF = new_list_rec_definition('ELIN_DEF',
  "(ELIN [] (x:a) = F) /\
  (ELIN (CONS h s) (x:a) = (x = h) \\/ (ELIN s a))";;

%-----%
% Some theorems about ELIN and other list operators %
%-----%
let NULL_NOT_ELIN = prove_thm('NULL_NOT_ELIN',
  "(1. NULL 1 ==> !x. ~(ELIN 1 x)",
  COND_ASSUME_TAC[NULL_EL]
  THEN GEN_TAC THEN STRIP_TAC THEN ASS_ASSUME_TAC[!];;

let ELIN_CONS = prove_thm('ELIN_CONS',
  "(1 x y. (ELIN 1 a) ==> (ELIN (CONS y 1) a)",
  LIST_INDUCT_TAC THEN ASSUME_TAC[ELIN_DEF:NOVA]
  THEN REPEAT GEN_TAC THEN STRIP_TAC THEN ASS_ASSUME_TAC[!];;

let ELIN_APPEND = prove_thm('ELIN_APPEND',
  "(1 12 a. (ELIN (APPEND 11 12) a) = ((ELIN 11 a) \\/ (ELIN 12 a))",
  LIST_INDUCT_TAC THEN ASSUME_TAC[APPEND:ELIN_DEF]
  THEN REPEAT GEN_TAC THEN ASS_ASSUME_TAC[DISJ_ASSOC]);;

let ELIN_EL = prove_thm('ELIN_EL',
  "(1:(c)list. a. ELIN 1 a ==> (?!n. a = EL n 1))",
  LIST_INDUCT_TAC THEN ASSUME_TAC[ELIN_DEF]
  THEN REPEAT STRIP_TAC THEN[
    EXISTS_TAC "0" THEN ASS_ASSUME_TAC[EL:NO]
    AND_TAC THEN EXISTS_TAC "SUC n" THEN ASS_ASSUME_TAC[EL:YJ]);;

%-----%
% Prove the equivalences of set membership (ID) and list membership (ELIN)%

```

```

let IS_ELEM = prove_thm('IS_ELEM',
  "is set. FINITE s ==> ?l::list. is. (s IS s) = (ELEM 1 s)",
  SET_INSERT_TAC THENL[
    REPEAT_TAC "[1] = list" THEN REWRITE_TAC[IS_ELEM_DEF,SET_IS_DEF];
    REPEAT_TAC "(CONS s (el. 1a. s IS s = ELEM 1 s))::= list"
    THEN REWRITE_TAC[IS_ELEM_DEF,IS_INSERT];
    THEN SRS_TAC THEN EQ_TAC THEN STRIP_TAC THENL[
      DISJ1_TAC THEN FIRST_ASSUM ACCEPT_TAC;
      DISJ2_TAC THEN FIRST_ASSUM (ASSUME_TAC < SELECT_RULE)
      THEN UNIQUE_TAC "(s el) IS s"
      THEN FIRST_ASSUM (\s. MATCH_ACCEPT_TAC
        ((set s EQ_IMP_RULE s SPEC_ALL) s));
      DISJ1_TAC THEN FIRST_ASSUM ACCEPT_TAC;
      DISJ2_TAC THEN FIRST_ASSUM (SUBST1_TAC < SPEC_ALL < SELECT_RULE)
      THEN FIRST_ASSUM ACCEPT_TAC];];

%-----%
% UNIQUE_EL is true if all elements of the list are distinct -%
%-----%
let UNIQUE_EL_DEF = new_list_rec_definition('UNIQUE_EL_DEF',
  "(UNIQUE_EL [] = ?) /\
  (UNIQUE_EL (CONS (hd::t) tl) =
  (EVERY (\s. ~(s = hd)) tl) /\ (UNIQUE_EL tl))");;

let UNIQUE_EL_TH = prove_thm('UNIQUE_EL_TH',
  "?l (h::t). UNIQUE_EL (CONS h l) ==> UNIQUE_EL l",
  SRS_REWRITE_TAC[UNIQUE_EL_DEF] THEN REWRITE_TAC[AND2_TH]);;

let UNIQUE_EL_SMP = prove_thm('UNIQUE_EL_SMP',
  "is s. UNIQUE_EL s",
  SRS_TAC THEN REWRITE_TAC[UNIQUE_EL_DEF,EVERY_DEF]);;

let ELEM_NOT_UNIQUE_EL_CONS = prove_thm('ELEM_NOT_UNIQUE_EL_CONS',
  "?l (h::t). (ELEM 1 h) ==> ~(UNIQUE_EL (CONS h l))",
  LIST_INSERT_TAC THENL[
    REWRITE_TAC[IS_ELEM_DEF];
    SRS_REWRITE_TAC[IS_ELEM_DEF,UNIQUE_EL_DEF]
    THEN REPEAT SRS_TAC THEN STRIP_TAC
    THEN SRS_REWRITE_TAC[EVERY_DEF] THENL[
      CORV_TAC (SRS_DEPTH_CONV REW_CONV)
      THEN SRS_REWRITE_TAC[IS_NORMAL_TH];
      SRS_THEN MP_TAC THEN REWRITE_TAC[IS_NORMAL_TH,UNIQUE_EL_DEF]
      THEN STRIP_TAC THEN SRS_REWRITE_TAC[[]];];];

let NOT_ELEM_UNIQUE_EL_CONS = prove_thm('NOT_ELEM_UNIQUE_EL_CONS',
  "?l (h::t). (UNIQUE_EL l) /\ ~(ELEM 1 h) ==> (UNIQUE_EL (CONS h l))",
  LIST_INSERT_TAC THENL[
    REWRITE_TAC[IS_ELEM_DEF,UNIQUE_EL_DEF,EVERY_DEF];
    PURR_SRS_REWRITE_TAC[IS_ELEM_DEF,UNIQUE_EL_DEF]
    THEN PURR_SRS_REWRITE_TAC[IS_NORMAL_TH,UNIQUE_EL_DEF,EVERY_DEF]
    THEN REPEAT SRS_TAC THEN STRIP_TAC
    THEN CORV_TAC (SRS_DEPTH_CONV REW_CONV) THEN SRS_TAC
    THEN POP_ASSUM (\s. STRIP_ASSUME_TAC
      (SRS_REWRITE_RULE[UNIQUE_EL_DEF] s))
    THEN SRS_REWRITE_TAC[] THEN CORV_TAC (SRS_DEPTH_CONV STR_CONV)
    THEN FIRST_ASSUM ACCEPT_TAC];];

```

```

-----
% ML_SET constructs a set containing all elements of a list -%
-----
let ML_SET_DEF = new_list_rec_definition('ML_SET_DEF',
  "(ML_SET [] = {})" /\
  "(ML_SET (CONS hd tl :> list) = INSERT hd (ML_SET tl))";);

let ML_SET_APPEND = prove_thm('ML_SET_APPEND',
  "(!l :> list) IS ML_SET (APPEND l l2) =
    (ML_SET l1) UNION (ML_SET l2)",
  LIST_INDUCT_TAC THEN[
    ASSUME_TAC[APPEND:ML_SET_DEF:UNION_EMPTY];
    SOME_ASSUME_TAC[APPEND:ML_SET_DEF];
    THEN SOME_ASSUME_TAC[ML_SET_DEF] THEN ASS_ASSUME_TAC[INSERT_UNION];
    THEN REPEAT GEN_TAC THEN COND_CASES_TAC THEN[
      POP_ASSUM (ASSUME_TAC <= ASSUME_RULE[]);
      THEN ASSUME_TAC (SPEC "(ML_SET l2) = {}" <=);
      (SPEC "(ML_SET l1) = {}" <=) (SPEC "(h :>) IS_IMP_IN_UNION)];
    THEN REPEAT GEN_TAC THEN IMP_RES_TAC ASSORTION;
    REPEAT_TAC[]];);

let ELLEN_IS_ML_SET = prove_thm('ELLEN_IS_ML_SET',
  "!x. ELLEN l x = x IS (ML_SET l)",
  LIST_INDUCT_TAC THEN[
    ASSUME_TAC[ELLEN_DEF:ML_SET_DEF:NOT_IN_EMPTY];
    ASSUME_TAC[ELLEN_DEF:ML_SET_DEF:IS_INSERT];
    THEN REPEAT GEN_TAC THEN ASS_ASSUME_TAC[]];);

-----
% DISJ_LIST --- two lists are disjoint if the sets of their elements %
% are disjoint %
-----
let DISJ_LIST_DEF = new_definition('DISJ_LIST_DEF',
  "DISJ_LIST (l1 :> (o)list) l2 = DISJOINT (ML_SET l1) (ML_SET l2)";);

let DISJ_LIST_EMPTY = prove_thm('DISJ_LIST_EMPTY',
  "!l :> (o)list. (DISJ_LIST [] l) /\ (DISJ_LIST l [])",
  ASSUME_TAC[DISJ_LIST_DEF:ML_SET_DEF:DISJOINT_DEF:INTER_EMPTY]););

let DISJ_LIST_CONS = prove_thm('DISJ_LIST_CONS',
  "!l1 (l2 :> (o)list) b. (DISJ_LIST (CONS b l1) l2) =
    ((DISJ_LIST l1 l2) /\ ~(ELLEN l2 b))",
  LIST_INDUCT_TAC THEN
  ASSUME_TAC[DISJ_LIST_DEF:ML_SET_DEF:ELLEN_IS_ML_SET:DISJOINT_INSERT]););

let DISJ_LIST_APPEND = prove_thm('DISJ_LIST_APPEND',
  "!l1 (l2 :> (o)list) l3. (DISJ_LIST (APPEND l1 l2) l3) =
    ((DISJ_LIST l1 l3) /\ (DISJ_LIST l2 l3))",
  LIST_INDUCT_TAC
  THEN SOME_ASSUME_TAC[APPEND:DISJ_LIST_CONS:DISJ_LIST_EMPTY];
  THEN REPEAT GEN_TAC THEN ASS_ASSUME_TAC[]
  THEN EQ_TAC THEN STRIP_TAC THEN ASS_ASSUME_TAC[]];);

let DISJ_LIST_CONS = prove_thm('DISJ_LIST_CONS',
  "(!l1 (o)list) IS. DISJ_LIST l1 l2 = DISJ_LIST l2 l1",

```

```

REWRITE_TAC[DISJ_LIST_DEF] THEN MATCH_ACCEPT_TAC DISJLIST_STM;;

-----
% Given a path p, VER_LIST returns a list of all vertices p visits -%
% V_L returns a list of all vertices except the initial one. -%
-----
let V_L_DEF = new_list_rec_definition('V_L_DEF',
  '(V_L [] = []) /\
   (V_L (CONS (hd "Edge") tl) = CONS (a_doe hd) (V_L tl))";);

let VER_LIST_DEF = new_list_rec_definition('VER_LIST_DEF',
  '(VER_LIST [] = []) /\
   (VER_LIST (CONS (hd "Edge") tl) = CONS (a_src hd) (V_L (CONS hd tl)))));

let V_L_APPEND = prove_thm('V_L_APPEND',
  "(p1 (p2 ('Edge'))list).
   (V_L (APPEND p1 p2)) = (APPEND (V_L p1) (V_L p2))",
  LIST_INDUCT_TAC THEN SUCH_REWRITE_TAC[APPEND_V_L_DEF]
  THEN SUCH_REWRITE_TAC[APPEND_V_L_DEF] THENL[
    GEN_TAC THEN REPT_TAC;
    SUCH_ASR_REWRITE_TAC[] THEN REPEAT GEN_TAC THEN REPT_TAC];);

let NOT_NULL_VER_LIST = prove_thm('NOT_NULL_VER_LIST',
  "(p ('Edge'))list.
   'NULL p => ((VER_LIST p) = (CONS (a_src (HD p)) (V_L p)))",
  GEN_TAC THEN STRIP_TAC
  THEN INP_RES_THEN (\th. SUCH_REWRITE_TAC[th])
  (SUCH_REWRITE_RULE[RE_STM_RQ] CONS)
  THEN REWRITE_TAC[VER_LIST_DEF, HD:TL_V_L_DEF];);

let VER_LIST_CONS = prove_thm('VER_LIST_CONS',
  "(p ('Edge')).
   (VER_LIST (CONS h p)) = (CONS (a_src h) (CONS (a_doe h) (V_L p)))",
  LIST_INDUCT_TAC THEN REWRITE_TAC[VER_LIST_DEF, V_L_DEF];);

let NOT_NULL_VER_LIST_CONS = prove_thm('NOT_NULL_VER_LIST_CONS',
  "'is th 'Edge'. 'NULL p /\ (a_doe h = a_src (HD p)) =>
   (VER_LIST (CONS h p) = CONS (a_src h) (VER_LIST p))",
  REPEAT STRIP_TAC THEN INP_RES_THEN (SUCH_A_TAC NOT_NULL_VER_LIST
  THEN ASR_REWRITE_TAC [VER_LIST_CONS]););

let TL_VER_LIST = prove_thm('TL_VER_LIST',
  "(p ('Edge'))list. 'NULL p => ((TL (VER_LIST p)) = (V_L p))",
  GEN_TAC THEN STRIP_TAC
  THEN INP_RES_THEN (\th. SUCH_REWRITE_TAC[th]) NOT_NULL_VER_LIST
  THEN REWRITE_TAC[TL];);

let VER_LIST_APPEND = prove_thm('VER_LIST_APPEND',
  "(p1 ('Edge'))list p2. 'NULL p1 /\ 'NULL p2 =>
   ((VER_LIST (APPEND p1 p2)) = (APPEND (VER_LIST p1) (TL (VER_LIST p2))))",
  LIST_INDUCT_TAC THENL[
    REWRITE_TAC[NULL];
    REPEAT GEN_TAC THEN STRIP_TAC THEN REWRITE_TAC[VER_LIST_DEF, APPEND]
    THEN INP_RES_THEN (\th. REWRITE_TAC[th]) TL_VER_LIST
    THEN SUCH_REWRITE_TAC[(SUCH_REWRITE_RULE[RE_STM_RQ] V_L_APPEND)]
    THEN REWRITE_TAC[APPEND];);

```

```

let UNIQUE_EL_CONS = prove.thm('UNIQUE_EL_CONS',
  "!(h:α). (UNIQUE_EL h) /\ ~(h IS (EL_SET h)) => (UNIQUE_EL (CONS h h))",
  LIST_INSERT_TAC THEN!
  REWRITE_TAC[EL_SET_DEF; UNIQUE_EL_DEF; SET_IS_EMPTY; EVENT_DEF];
  CONC_REWRITE_TAC[UNIQUE_EL_DEF]
  THEN REWRITE_TAC[EL_SET_DEF; IS_INSERT; IS_NORMAL_THM; EVENT_DEF]
  THEN REPEAT STRIP_TAC THEN!
    REW_TAC THEN CHV_TAC (CONC_DEPTH_CONV STR_CONV)
    THEN ASS_REWRITE_TAC[];
    AND THEN RP_TAC THEN REWRITE_TAC[UNIQUE_EL_DEF]
    THEN REPEAT STRIP_TAC THEN REW_TAC
    ASS_REWRITE_TAC[UNIQUE_EL_DEF]]];

let NOT_UNIQUE_EL_CONS = prove.thm('NOT_UNIQUE_EL_CONS',
  "!(h:α). (h IS (EL_SET h)) => (UNIQUE_EL (CONS h h) = F)",
  REWRITE_TAC[UNIQUE_EL_DEF; IS_NORMAL_THM]
  THEN LIST_INSERT_TAC THEN!
  REWRITE_TAC[EL_SET_DEF; UNIQUE_EL_DEF; SET_IS_EMPTY];
  CONC_REWRITE_TAC[EL_SET_DEF]
  THEN REWRITE_TAC[UNIQUE_EL_DEF]
  THEN REWRITE_TAC[IS_INSERT]
  THEN REPEAT GEN_TAC THEN STRIP_TAC THEN!
    REWRITE_TAC[EVENT_DEF] THEN REW_TAC THEN ASS_REWRITE_TAC[],
    REWRITE_TAC[EVENT_DEF; IS_NORMAL_THM]
    THEN REW_TAC THEN ASS_REWRITE_TAC[]];

let UNIQUE_EL_APPEND =
  let thm = "(!l1 (l2:α list). UNIQUE_EL (APPEND l1 l2) =
    (UNIQUE_EL l1) /\ (UNIQUE_EL l2) /\ (DISJ_LIST l1 l2))"
  in
  let EVERY_APPEND = TAC_PROOF([],
    "(!l1:α list) l2 P.
    EVERY P (APPEND l1 l2) = (EVERY P l1) /\ (EVERY P l2)",
    LIST_INSERT_TAC THEN
    ASS_REWRITE_TAC[APPEND; EVERY_DEF; CONC; ANDRO])
  in
  let lem = TAC_PROOF([],
    "!(l:(α)list) h. EVENT(h. `(h = h)) = `(ELM h h)",
    LIST_INSERT_TAC THEN REWRITE_TAC[EVERY_DEF; ELEM_DEF]
    THEN CHV_TAC (CONC_DEPTH_CONV STR_CONV) THEN REPEAT GEN_TAC
    THEN PURS_CONC ASS_REWRITE_TAC[IS_NORMAL_THM]
    THEN EQ_TAC THEN STRIP_TAC THEN CHV_TAC (CONC_DEPTH_CONV STR_CONV)
    THEN OBJ1_TAC THEN FIRST_ASSUM ACCEPT_TAC)
  in
  prove.thm('UNIQUE_EL_APPEND', thm,
    LIST_INSERT_TAC THEN!
    REWRITE_TAC[APPEND; EL_SET_DEF; UNIQUE_EL_DEF; DISJ_LIST_DEF;
    DISJOINT_DEF; IS_EMPTY];
    CONC_REWRITE_TAC[APPEND; EL_SET_DEF]
    THEN CONC_REWRITE_TAC[UNIQUE_EL_DEF]
    THEN PURS_CONC ASS_REWRITE_TAC[DISJ_LIST_CONS; EVERY_APPEND]
    THEN PURS_CONC ASS_REWRITE_TAC[lem] THEN REPEAT GEN_TAC
    THEN EQ_TAC THEN STRIP_TAC THEN ASS_REWRITE_TAC[]];

let UNIQUE_V_L_CONS = prove.thm('UNIQUE_V_L_CONS',
  "(!p:('Rdef)list) h. UNIQUE_EL(V_L p) /\ ~(ELM (V_L p) (e_def h))
  => UNIQUE_EL(V_L (CONS h p))",

```



```

REWRITE_TAC[V_L_DEF]
THEN MATCH_ACCEPT_TAC NOT_ELSE_UNIQUE_EL_CASES;;

let UNIQUE_VER_LIST_CASE = prove_thm('UNIQUE_VER_LIST_CASE',
  "(p:('Edge)list) b ~ (BULL p) /\
  UNIQUE_EL(VER_LIST p) /\ (a_src (HD p)) = (a_dst b)) /\
  ~(LOOP b) /\ ~(ELSN (VER_LIST p) (a_src b)) ==>
  UNIQUE_EL(VER_LIST (CONS b p))",
  PURE_ONCE_REWRITING_TAC[VER_LIST_DEF] THEN REPEAT STRIP_TAC
  THEN IMP_RES_TAC NOT_ELSE_UNIQUE_EL_CASE
  THEN POP_ASSUM MP_TAC THEN IMP_RES THEN SUBST1_TAC NOT_BULL_VER_LIST
  THEN ASS_REWRITE_TAC[V_L_DEF]::);;

let UNIQUE_EL_VER_LIST_TL = prove_thm('UNIQUE_EL_VER_LIST_TL',
  "!(p:('Edge)list). ~(BULL p) ==>
  UNIQUE_EL (VER_LIST p) ==> UNIQUE_EL (TL (VER_LIST p))",
  GEN_TAC THEN STRIP_TAC
  THEN IMP_RES THEN (\b. REWRITE_TAC[tl]) NOT_BULL_VER_LIST
  THEN REWRITE_TAC[UNIQUE_EL_DEF;TL_DEF;NOT_DEF]::);;

let UNIQUE_VER_LIST_APPEND = prove_thm('UNIQUE_VER_LIST_APPEND',
  "(p1:('Edge)list) p2 (G:('Graph) * BULL p1 /\ 'BULL p2 ==>
  UNIQUE_EL(VER_LIST p1) /\ UNIQUE_EL(VER_LIST p2) /\
  BULL_LIST(V_L p1)(V_L p2) /\ ~(ELSN (VER_LIST p2) (a_src (HD p1)))
  ==> UNIQUE_EL(VER_LIST(APPEND p1 p2))",
  REPEAT GEN_TAC THEN STRIP_TAC
  THEN IMP_RES_TAC VER_LIST_APPEND THEN STRIP_TAC
  THEN ASS_REWRITE_TAC[UNIQUE_EL_APPEND]
  THEN CURS_TAC THEN[
    IMP_RES_TAC UNIQUE_EL_VER_LIST_TL
    IMP_RES THEN SUBST1_TAC TL_VER_LIST
    THEN UNDISCH_TAC "~(ELSN(VER_LIST(p2:('Edge)list))(a_src(HD(p1:('Edge)list))))"
    THEN IMP_RES THEN SUBST1_TAC NOT_BULL_VER_LIST
    THEN PURE_ONCE_REWRITING_TAC[DISJ_LIST_CASE;ELSN_DEF]
    THEN PURE_ONCE_REWRITE_TAC[MS_NORMAR_THM]
    THEN STRIP_TAC THEN ASS_REWRITE_TAC[]];;

close_theory();;

```

B.5 The file mk_path.ml

```

-----
% FILE: mk_path.ml
% DESCRIPTION: definition of paths and some theorems
% AUTHOR: Mai Hong DATE: 1 AUG 1990 modified Jan 91
%-----
new_theory'path';

load_library'auto';
new_parent'graph';;
autolabel_all'graph';;
new_parent'list';;
autolabel_all'list';;

```

```

-----
% Vertex, Edge and Graph are defined as abbreviations for the types used
% to represent vertices, edges and graphs.
-----
let Vertex = "v" and
    Edge = "(e: b < a < a)" and
    Graph = "(g: (a) set b < (a < a < a) set a)";

let HD_APPEND = TAC_PROOF([],
    "(pi p2: (e) list. ('NULL p1) ==> (HD (APPEND p1 p2) = HD p1)",
    LIST_ISSUES_TAC THEN
    DERIVATIVE_TAC THEN
    DERIVATIVE_TAC[APPEND; HD];););

%-----
% A walk in a graph is a list of edges in which the KEY of each element,
% except the last, is equal to the KEY of the following element.
%-----
let WALK_TAIL_DEF = new_list_rec_definition('WALK_TAIL_DEF',
    "(hd: 'Edge) tl. WALK_TAIL (CONS hd tl) a =
    (GRAPH a) /\ (hd IN_EDGE a) /\
    (('NULL tl) /\ (WALK_TAIL tl a) /\ (a_dom hd = a_arc (HD tl)))");;

let WALK_DEF = new_definition('WALK_DEF',
    "WALK a (a: ('Edge) list) = ('NULL a) /\ (WALK_TAIL a a)");;

let WALK_KEYTY_DEF = new_definition('WALK_KEYTY_DEF',
    "WALK_KEYTY (l: ('Edge) list) = a_arc (HD l)");;

let WALK_KEYTY_DEF = new_list_rec_definition('WALK_KEYTY_DEF',
    "WALK_KEYTY (CONS (hd: 'Edge) tl) =
    ('NULL tl) ==> (a_dom hd) | (WALK_KEYTY tl)");;

%-----
% A trail in a graph is a walk whose edges are all distinct
%-----
let TRAIL_DEF = new_definition('TRAIL_DEF',
    "TRAIL (g: 'Graph) (l: ('Edge) list) = (WALK g l) /\ (UNIQUE_EL l)");;

%-----
% A path in a graph is a trail whose vertices are all distinct
%-----
let PATH_DEF = new_definition('PATH_DEF',
    "PATH (g: 'Graph) (l: ('Edge) list) =
    (TRAIL g l) /\ (UNIQUE_EL (VEX_LIST l))");;

let PATH_KEYTY_DEF = new_definition('PATH_KEYTY_DEF',
    "PATH_KEYTY (l: ('Edge) list) = a_arc (HD l)");;

let PATH_KEYTY_DEF = new_definition('PATH_KEYTY_DEF',
    "PATH_KEYTY (p: ('Edge) list) = WALK_KEYTY p");;

let KEY_NULL_LIST = TAC_PROOF([],
    "(l: (a) list. 'NULL l ==> ('th a. l = (CONS b a)))",
    LIST_ISSUES_TAC THEN[ DERIVATIVE_TAC[NULL];
    DERIVATIVE_TAC THEN EXISTS_TAC "b: a"

```

```

THEN EXISTS_TAC "1:(<0)list" THEN REFL_TAC];];

%-----
% Some facts about WALK, TRAIL and PATH
%-----
let PATH_TRAIL = prove.thm('PATH_TRAIL',
  "(1:(<'Edge)list) G. PATH G 1 ==> TRAIL G 1",
  REWRITE_TAC[PATH_DEF;AND1_THM]);];

let TRAIL_WALK = prove.thm('TRAIL_WALK',
  "(1:(<'Edge)list) G. TRAIL G 1 ==> WALK G 1",
  REWRITE_TAC[TRAIL_DEF;AND1_THM]);];

let PATH_WALK = prove.thm('PATH_WALK',
  "(1:(<'Edge)list) G. PATH G 1 ==> WALK G 1",
  REPEAT STRIP_TAC THEN INP_RES_TAC PATH_TRAIL THEN INP_RES_TAC TRAIL_WALK);];

let PATH_GRAPH = prove.thm('PATH_GRAPH',
  "(1:(G:Graph) 1. PATH G 1 ==> GRAPH G",
  PUSH_REWRITE_TAC[PATH_DEF;TRAIL_DEF;WALK_DEF]
  THEN REPEAT STRIP_TAC THEN UNDISCH_TAC "WALK_TAIL 1 (G:'Graph)"
  THEN INP_RES_TAC GET_WALK_LIST THEN POP_ASSUM SUBST1_TAC
  THEN PUSH_ONCE REWRITE_TAC[WALK_TAIL_DEF]
  THEN STRIP_TAC THEN FIRST_ASSUM ACCEPT_TAC);];

let PATH_GET_WALK = prove.thm('PATH_GET_WALK',
  "(p:(G:'Graph) PATH G 1 ==> 'WALK 1",
  REPEAT GET_TAC THEN DISCH_TAC THEN INP_RES_TAC THEN SP_TAC PATH_WALK
  THEN REWRITE_TAC[WALK_DEF] THEN STRIP_TAC);];

let PATH_WALK_ENTRY = prove.thm('PATH_WALK_ENTRY',
  "(p:(<'Edge)list. PATH_ENTRY p = WALK_ENTRY p",
  REWRITE_TAC[PATH_ENTRY_DEF;WALK_ENTRY_DEF]);];

let PATH_CONNECTED = prove.thm('PATH_CONNECTED',
  "(p:(h:'Edge) d.
  (PATH G (CONS h p)) /\ ~(NULL p) ==> ((e,das h) = (e,arc (RD p))))",
  REWRITE_TAC[PATH_DEF;TRAIL_DEF;WALK_DEF]
  THEN PUSH_REWRITE_TAC[WALK_TAIL_DEF] THEN REPEAT GET_TAC THEN STRIP_TAC
  THEN UNDISCH_TAC "~ NULL (p:(<'Edge)list)" THEN ASS_REWRITE_TAC[]);];

%-----
% A graph is connected if there is a path in it between any pair
% of vertices
%-----
let CONNECTED_DEF = new_definition('CONNECTED_DEF',
  "CONNECTED (G:'Graph) =
  (GRAPH G) /\
  (forall v2. (v1 IN VERTEX G) /\ (v2 IN VERTEX G) /\ (v1 = v2) ==>
  (exists (PATH G 1) /\ (v1 = PATH_ENTRY 1) /\ (v2 = PATH_EXIT 1)))");];

let CONNECTED_GRAPH = prove.thm('CONNECTED_GRAPH',
  "(1:(G:'Graph). CONNECTED G ==> GRAPH G",
  PUSH_ONCE REWRITE_TAC[CONNECTED_DEF]
  THEN REPEAT STRIP_TAC THEN FIRST_ASSUM ACCEPT_TAC);];

let CONNECTED_SIZE = prove.thm('CONNECTED_SIZE',

```

```

"to CONNECTED ((v), {}):"Graph";
REWRITE_TAC[CONNECTED_DEF;IS_VERTEX_DEF;GRAPH_DEF;VERTICES_DEF;NOT_IS_EMPTY;IS_INSERT]
THEN REPEAT GEN_TAC THEN STRIP_TAC THEN POP_ASSUM MP_TAC THEN ASS_REWRITE_TAC[]];

-----X
X- Two paths are disjoint iff their edge sets are disjoint and their
X- vertex sets except the enrtion are disjoint.
X-----X
let DISJ_PATH_DEF = new_definition('DISJ_PATH_DEF',
  "DISJ_PATH (G:('Graph)) (p1:('EdgeList)) (p2:('EdgeList)) =
  (PATH (G p1) /\ (PATH (G p2) /\ (DISJ_LIST p1 p2) /\
  (DISJ_LIST (V.L p1) (V.L p2)))");

-----X
X- HAS_PATH (G v1 v2) if there is a path in G from v1 to v2
X-----X
let HAS_PATH_DEF = new_definition('HAS_PATH_DEF',
  "HAS_PATH (G:('Graph)) v1 v2 = ?p:('EdgeList). PATH (G p) /\
  (PATH_ENTRY p = v1) /\ (PATH_EXIT p = v2)");

let WALK_ENTRY_COST = prove_thm('WALK_ENTRY_COST',
  "(?p:('EdgeList)) h G. (WALK_ENTRY (COSTS h p)) = (a,arc h)",
  REWRITE_TAC[WALK_ENTRY_DEF;H]);

let WALK_ENTRY_APPEND = prove_thm('WALK_ENTRY_APPEND',
  "?p1 (?p2:('EdgeList)) G. (WALK (G p1) /\ (WALK (G p2)) ==>
  (WALK_ENTRY (APPEND p1 p2) = WALK_ENTRY p1)",
  REWRITE_TAC[WALK_DEF;WALK_ENTRY_DEF] THEN REPEAT GEN_TAC
  THEN STRIP_TAC THEN IMP_RES_TAC HD_APPEND THEN ASS_REWRITE_TAC[]];

let WALK_EXIT_APPEND_lemma =
  let NOT_NULL_APPEND = TAC_PROOF([],
    "(?p1 (?p2:('EdgeList)) p2. "NULL p2 ==> "NULL (APPEND p1 p2)",
    LIST_INSERT_TAC THEN (REWRITE_TAC[APPEND;NULL]) in
    TAC_PROOF([[]],
      "?p1 (?p2:('EdgeList)). "NULL p2 ==>
      (WALK_EXIT (APPEND p1 p2) = WALK_EXIT p2)",
      LIST_INSERT_TAC THEN REWRITE_TAC[APPEND]
      THEN GEN_REWRITE_TAC[WALK;WALK_EXIT_DEF] THEN REPEAT STRIP_TAC THEN ASS_TAC
      THEN IMP_RES_TAC NOT_NULL_APPEND THEN ASS_REWRITE_TAC[]];
  let WALK_EXIT_APPEND = prove_thm('WALK_EXIT_APPEND',
    "?p1 (?p2:('EdgeList)) G. (WALK (G p1) /\ (WALK (G p2)) ==>
    (WALK_EXIT (APPEND p1 p2) = WALK_EXIT p2)",
    REWRITE_TAC[WALK_DEF] THEN REPEAT STRIP_TAC
    THEN IMP_RES_TAC WALK_EXIT_APPEND_lemma THEN ASS_REWRITE_TAC[]];

let PATH_ENTRY_SIMP = prove_thm('PATH_ENTRY_SIMP',
  "(?u,v) w (u==v). PATH_ENTRY(u,v,w) = w",
  REWRITE_TAC[PATH_ENTRY_DEF;a,arc;HD]);

let PATH_EXIT_SIMP = prove_thm('PATH_EXIT_SIMP',
  "(?u,v) w (u==v). PATH_EXIT(u,v,w) = w",
  REWRITE_TAC[PATH_EXIT_DEF;WALK_EXIT_DEF;a,arc;HD;NULL]);

let PATH_ENTRY_COST = prove_thm('PATH_ENTRY_COST',
  "?p (h:('Edge)). PATH_ENTRY(COSTS h p) = a,arc h",

```

```

PUSH_ONCE_REMOVE_TAC[PATH_WALK_ENTRY]
THEN MATCH_ACCORDY_TAC WALK_ENTRY.CODES::

let PATH_EXIT.CODES = prove_thm('PATH_EXIT.CODES',
  "ip h ('Edge'):  WALK p ==> (PATH_EXIT.CODES h p) = PATH_EXIT p",
  PUSH_REMOVE_TAC[PATH_EXIT.BODY, WALK_EXIT.BODY]
  THEN REPEAT GEN_TAC THEN DISCH_TAC THEN ASS_REMOVE_TAC[]::)

let PATH_ENTRY.APPENDS = prove_thm('PATH_ENTRY.APPENDS',
  "!! (L2:('Edge'list): PATHS 0 L1 ==>
    (PATH_ENTRY.APPENDS L1 L2) = PATH_ENTRY L1)",
  REPEAT STRIP_TAC THEN IMP_RES_TAC PATH_BUTL
  THEN IMP_RES_TAC BUTL_LIST THEN POP_ASSUM SOME01_TAC
  THEN REMOVE_TAC[APPENDS:PATH_ENTRY.BODY,HD0]::)

let IMP_APPEND_BUTL_BUTL = TAC_PROOF([[]],
  "!! (L2:('Edge'list):  BUTL L1 V / BUTL L2 ==> 'BUTL (APPEND L1 L2)"),
  LIST_INSERT_TAC THEN REMOVE_TAC[BUTL,APPEND]::)

let PATH_EXIT.APPENDS = prove_thm('PATH_EXIT.APPENDS',
  "!! (L2:('Edge'list):  WALK L2 ==>
    (PATH_EXIT.APPENDS L1 L2) = PATH_EXIT L2)",
  PUSH_ONCE_REMOVE_TAC[PATH_EXIT.BODY] THEN
  LIST_INSERT_TAC THEN REMOVE_TAC[APPENDS:WALK_EXIT.BODY]
  THEN REPEAT STRIP_TAC THEN IMP_RES_TAC IMP_APPEND_BUTL_BUTL
  THEN ASS_REMOVE_TAC[] THEN RES_TAC::)

let WALK.CODES = prove_thm('WALK.CODES',
  "ip h ('Graph') :
    (WALK 0 p) /\ (h 10.HD0 0) /\ ((c,den h) = WALK_ENTRY p) ==>
    (WALK 0 (CODES h p))",
  REMOVE_TAC[WALK.WALK_BODY:WALK_ENTRY.BODY]
  THEN LIST_INSERT_TAC THENL[
    REMOVE_TAC[NULL],
    ORCH_REMOVE_TAC[WALK.TAIL_BODY] THEN REMOVE_TAC[NULL,HD]
  ] THEN REPEAT GEN_TAC THEN STRIP_TAC THENL[
    PUSH_ONCE_REMOVE_TAC[WALK.TAIL_BODY],
    ASS_CASES_TAC "WALK (p:('Edge'list))" THENL[
      PUSH_ONCE_REMOVE_TAC[WALK.TAIL_BODY], RES_TAC[]
    ] THEN ASS_REMOVE_TAC[]::)

let WALK.APPENDS = prove_thm('WALK.APPENDS',
  "ip p2 ('Graph') :
    (WALK 0 p1) /\ (WALK 0 p2) /\ (WALK_EXIT p1 = WALK_ENTRY p2) ==>
    (WALK 0 (APPENDS p1 p2))",
  REMOVE_TAC[WALK.BODY] THEN LIST_INSERT_TAC THEN REMOVE_TAC[APPENDS:NULL]
  THEN ORCH_REMOVE_TAC[WALK.TAIL_BODY] THEN ORCH_REMOVE_TAC[WALK_EXIT_BODY]
  THEN ASS_CASES_TAC "WALK (p1:('Edge'list))" THENL[
    IMP_RES_TAC NULL_BUTL
    THEN ASS_REMOVE_TAC[APPENDS:NULL,WALK_ENTRY_BODY]
    THEN REPEAT GEN_TAC THEN STRIP_TAC THEN ASS_REMOVE_TAC[]
    THEN ASS_REMOVE_TAC[] THEN REPEAT GEN_TAC THEN STRIP_TAC
    THEN RES_TAC THEN IMP_RES_TAC HD_APPEND THEN ASS_REMOVE_TAC[]::)

let WALK.CAT = prove_thm('WALK.CAT',
  "!(G:'Graph') p1 p2. (WALK 0 p1) /\ (WALK 0 p2) /\
    (WALK_EXIT p1 = (WALK_ENTRY p2):0) ==>
```

```

    Tp3. (WALK e p3) /\ (WALK_EXIT p3 = WALK_EXIT p1) /\
      (WALK_EXIT p3 = WALK_EXIT p1) /\ (p3 = APPEND p1 p2)";
  REPEAT ONE_TAC THEN STRIP_TAC THEN EXISTS_TAC "APPEND p1 (p3:('Edge::list))"
  THEN CHU1_TAC THEN [
    IMP_RES_TAC WALK_APPEND;
    IMP_RES_TAC WALK_EXIT_APPEND
  ] THEN IMP_RES_TAC WALK_EXIT_APPEND THEN ASS_REWRITE_TAC[]];

let PATH_EDGE_EG_LOOP = prove(thm('PATH_EDGE_EG_LOOP',
  "!(g:('Edge::t)) e. (PATH e (CONN h p1)) => (('e,src h = e,des h))",
  REWRITE_TAC[PATH_RES_TRAIL_DEF;WALK_DEF]
  THEN ONCE_REWRITE_TAC[WALK_TAIL_DEF;UNIQUE_EL_DEF;VER_LIST_DEF]
  THEN ONCE_REWRITE_TAC[V.L_DEF] THEN ONCE_REWRITE_TAC[UNIQUE_EL_DEF]
  THEN ONCE_REWRITE_TAC[EVENT_DEF] THEN META_TAC
  THEN REPEAT ONE_TAC THEN STRIP_TAC
  THEN CHU1_TAC [ONCE_DEPTH_CONV (REWRITE_CONV EG_SYM_DEF)]
  THEN ASS_REWRITE_TAC[]]);

let PATH_STRIP = prove(thm('PATH_STRIP',
  "!(g:('Graph::t)) e. (GRAPH g) /\ (e IS_EDGE g) /\ ~(LOOP e) => (PATH e [e])",
  PURS_REWRITE_TAC[PATH_RES_TRAIL_DEF;WALK_DEF;WALK_TAIL_DEF]
  THEN REWRITE_TAC[BULL_DEF;UNIQUE_EL_DEF;VER_LIST_DEF;V.L_DEF;EVENT_DEF;LOOP_DEF]
  THEN CHU1_TAC [ONCE_DEPTH_CONV META_CONV]
  THEN REPEAT ONE_TAC THEN STRIP_TAC
  THEN CHU1_TAC [ONCE_DEPTH_CONV EG_SYM_DEF]
  THEN REPEAT CHU1_TAC THEN FIRST_ASSUM ACCEPT_TAC]);

let PATH_CONN = prove(thm('PATH_CONN',
  "!(p h (t:('Graph::t)))
  (GRAPH t) /\ (PATH e p) /\ (h IS_EDGE t) /\
  ((PATH_EXIT p) = (e,des h)) /\
  ~(LOOP e) /\ ~(ELIM (VER_LIST p) (e,src h)) /\
  ~(ELIM p h) => (PATH e (CONN h p1))",
  LIT1,INDUCT_TAC THEN [
    REPEAT STRIP_TAC THEN IMP_RES_TAC PATH_STRIP;
    REWRITE_TAC[PATH_RES_TRAIL_DEF;PATH_WALK_EXIT]
  ] THEN REPEAT STRIP_TAC THEN [
    IMP_RES_TAC (CHU1_BULK [ONCE_DEPTH_CONV EG_SYM_DEF] WALK_CONN);
    IMP_RES_TAC DEF_ELIM_UNIQUE_EL_CONN;
    MATCH_MP_TAC (UNIQUE_VER_LIST_CONN THEN PURS_ONCE_REWRITE_TAC[MB]
    THEN ASS_REWRITE_TAC[BULL])
    THEN FIRST_ASSUM (lambda ACCEP_TAC (REWRITE_RULE[WALK_EXIT_CONN] s))];];

let PATH_CAT =
  let thm =
    "!(p1 (p2:('Edge::list)) (GRAPH g) /\
    (DISJ_PATH e p1 p2) /\ (PATH_EXIT p1 = PATH_EXIT p2) /\
    ~(ELIM (VER_LIST p1) (PATH_EXIT p1)) =>
    (Tp3. (PATH e p3) /\ (PATH_EXIT p3 = PATH_EXIT p1) /\
    (PATH_EXIT p3 = PATH_EXIT p2) /\ (p3 = APPEND p1 p2))"
  in
    let leml = TAC_PROOF([],
      "!(p:('Edge::list)). PATH e p => UNIQUE_EL p",
      REWRITE_TAC[PATH_RES_TRAIL_DEF]
    ) THEN REPEAT STRIP_TAC THEN FIRST_ASSUM ACCEPT_TAC;
  in

```

```

let lem2 = PORN_CONV_REWRITE_RULE[
  (CONV_RULE (CONV_DEPTH_CONV SYM_CONV) WALK_ENTRY_DEF)]
  UNIQUE_VEB_LIST_APPEND

in
prove_thm('PATH_CAT', thm,
  PORN_REWRITE_TAC(PATH_DEF; TRAIL_DEF; B16); PATH_DEF; PATH_EXIT_DEF; PATH_WALK_ENTRY)
  THEN REPEAT STRIP_TAC THEN REWRITE_TAC "APPEND p1 (p2:('EdgeList))"
  THEN REPEAT CONJ_TAC THENL[
    IMP_RES_TAC PATH_WALK THEN IMP_RES_TAC WALK_APPEND;
    IMP_RES_TAC lem1 THEN IMP_RES_TAC UNIQUE_EL_APPEND;
    IMP_RES_TAC WALK_DEF THEN IMP_RES_TAC lem2;
    IMP_RES_TAC WALK_ENTRY_APPEND;
    IMP_RES_TAC WALK_EXIT_APPEND;
    REW_TAC[]];

let PATH_APPEND = prove_thm('PATH_APPEND',
  "is p1 (p2:('EdgeList)): (GRAPH G) /\
  (B12_PATH G p1 p2) /\ (PATH_EXIT p1 = PATH_ENTRY p2) /\
  'ELEM (VEB_LIST p2) (PATH_ENTRY p1) ==>
  (PATH G (APPEND p1 p2))",
  REPEAT STRIP_TAC THEN IMP_RES_TAC PATH_CAT
  THEN UNDISCH_TAC "PATH (G:('Graph)) p3" THEN ASS_REWRITE_TAC[]];

% |- is1 G2 v1 v2 n1 n2.
  GRAPH G1 /\ GRAPH G2 /\ v1 IS_VERTEX G1 /\ v2 IS_VERTEX G2 ==>
  GRAPH
  ((v1,v2,n1) INSERT_EDGE ((v2,v1,n2) INSERT_EDGE (G1 G_UNION G2))) %
let lemma1 = GEN_ALL (DISCH_ALL (CONJUNCT1 (SPDC_ALL
  (CONSTRN_ALL (SPDC_ALL G_UNION_IMP_EDGE)))));

let repeat n f =
  letrec rep n f l = if n = 0 then l else (f . (rep (n-1) f l)) in
  (rep n f []);

let PATH_EXIT_DEF = prove_thm('PATH_EXIT_DEF',
  "(G:('Graph)). (PATH G [])",
  REWRITE_TAC(PATH_DEF; TRAIL_DEF; WALK_DEF; NULL));

let WALK_TAIL_G_UNION = prove_thm('WALK_TAIL_G_UNION',
  "(!l:('EdgeList)) G1 G2.
  (GRAPH G1) /\ (GRAPH G2) /\ (WALK_TAIL 1 G1) ==>
  WALK_TAIL 1 (G1 G_UNION G2)",
  LIST_INDUCT_TAC THEN REWRITE_TAC(WALK_TAIL_DEF)
  THEN REPEAT GEN_TAC THEN STRIP_TAC THEN REPEAT CONJ_TAC THENL[
    IMP_RES_TAC GRAPH_UNION;
    ASS_REWRITE_TAC(INNER_IN_UNION);
    ASS_REWRITE_TAC[];
    IMP_RES_TAC GRAPH_UNION;
    ASS_REWRITE_TAC(INNER_IN_UNION);
    RES_TAC THEN ASS_REWRITE_TAC[]];

let PATH_G_UNION = prove_thm('PATH_G_UNION',
  "(!l:('EdgeList)) G1 G2. (GRAPH G1) /\ (GRAPH G2) /\
  (PATH G1 l) ==> PATH (G1 G_UNION G2) l",
  REWRITE_TAC(PATH_DEF; TRAIL_DEF; WALK_DEF) THEN LIST_INDUCT_TAC
  THEN REWRITE_TAC(NULL.WALK_TAIL_DEF) THEN REPEAT GEN_TAC THEN STRIP_TAC
  THEN REPEAT CONJ_TAC THEN ASS_REWRITE_TAC[] THENL[

```

```

IMP_RHS_TAC GRAPHS_UNION; ASSN_REWRITE_TAC[EDGE_IN_UNION];
IMP_RHS_TAC GRAPHS_UNION; ASSN_REWRITE_TAC[EDGE_IN_UNION];
IMP_RHS_TAC WALK_TAIL_0_UNION THEN ASSN_REWRITE_TAC[{}];

let WALK_TAIL_IS_VERTEX = prove_thm('WALK_TAIL_IS_VERTEX',
  "!(l:('Edge::list)) v 0. (WALK_TAIL 1 0) ==>
  (WALK_TAIL 1 (v INSERT_VERTEX 0))",
  LIST_INDUCT_TAC THEN REWRITE_TAC[WALK_TAIL_DEF]
  THEN REPEAT STRIP_TAC THEN [
    IMP_RHS THEN MATCH_ACCEPT_TAC GRAPHS_INSERT_VERTEX;
    IMP_RHS THEN MATCH_ACCEPT_TAC IS_INSERT_VERTEX;
    ASSN_REWRITE_TAC[{}];
    IMP_RHS THEN MATCH_ACCEPT_TAC GRAPHS_INSERT_VERTEX;
    IMP_RHS THEN MATCH_ACCEPT_TAC IS_INSERT_VERTEX;
    RES_TAC THEN ASSN_REWRITE_TAC[{}];];

let WALK_TAIL_IS_EDGE = prove_thm('WALK_TAIL_IS_EDGE',
  "!(l:('Edge::list)) e 0. (WALK_TAIL 1 0) ==> (WALK_TAIL 1 (e INSERT_EDGE 0))",
  LIST_INDUCT_TAC THEN REWRITE_TAC[WALK_TAIL_DEF]
  THEN REPEAT STRIP_TAC THEN [
    IMP_RHS THEN MATCH_ACCEPT_TAC GRAPHS_INSERT_EDGE;
    IMP_RHS THEN MATCH_ACCEPT_TAC IS_INSERT_EDGE;
    ASSN_REWRITE_TAC[{}];
    IMP_RHS THEN MATCH_ACCEPT_TAC GRAPHS_INSERT_EDGE;
    IMP_RHS THEN MATCH_ACCEPT_TAC IS_INSERT_EDGE;
    RES_TAC THEN ASSN_REWRITE_TAC[{}];];

let PATH_IS_VERTEX = prove_thm('PATH_IS_VERTEX',
  "!(l:('Edge::list)) v 0. (PATH 0 1) ==> (PATH (v INSERT_VERTEX 0) 1)",
  LIST_INDUCT_TAC THEN [
    REWRITE_TAC[PATH_DEF_01];
    REWRITE_TAC[PATH_DEF_TRAIL_DEF_WALK_DEF]
    THEN REPEAT RES_TAC THEN STRIP_TAC THEN ASSN_REWRITE_TAC[{}];
    THEN IMP_RHS THEN MATCH_ACCEPT_TAC WALK_TAIL_IS_VERTEX];];

let PATH_IS_EDGE = prove_thm('PATH_IS_EDGE',
  "!(l:('Edge::list)) e 0. (PATH 0 1) ==> (PATH (e INSERT_EDGE 0) 1)",
  LIST_INDUCT_TAC THEN [
    REWRITE_TAC[PATH_DEF_01];
    REWRITE_TAC[PATH_DEF_TRAIL_DEF_WALK_DEF]
    THEN REPEAT RES_TAC THEN STRIP_TAC THEN ASSN_REWRITE_TAC[{}];
    THEN IMP_RHS THEN MATCH_ACCEPT_TAC WALK_TAIL_IS_EDGE];];

let PATH_IS_EDGE2 = prove_thm('PATH_IS_EDGE2',
  "!(G:'Graph') v1 v2.
  (GRAPH G) /\ (v1 IS_VERTEX G) /\ (v2 IS_VERTEX G) /\ (v1 = v2) ==>
  (v. PATH (v1,v2,0) INSERT_EDGE 0) [(v1,v2,0)]",
  PURE_REWRITE_TAC[PATH_DEF_TRAIL_DEF_WALK_DEF]
  THEN REWRITE_TAC[WALK_TAIL_DEF; NULL]
  THEN REPEAT STRIP_TAC THEN [
    MATCH_MP_TAC GRAPHS_INSERT_EDGE THEN FIRST_ASSUM ACCEPT_TAC;
    IMP_RHS_TAC EDGE_IN_INSERT2 THEN FIRST_ASSUM MATCH_ACCEPT_TAC;
    MATCH_ACCEPT_TAC UNIQUE_EDGE_STRIP;
    PURE_REWRITE_TAC[VER_LIST_DEF_V_L_DEF_u_eq_u_def]
    THEN REWRITE_TAC[UNIQUE_EL_DEF; SUBSET_DEF]
    THEN RES_TAC THEN CHV_TAC (CHCH_DEPTH_CHE? STR_CONV)
    THEN FIRST_ASSUM ACCEPT_TAC];];

```



```

let PATH_IS_EDGE = prove,thm('PATH_IS_EDGE',
  "({(g:graph) & 1. PATH g (CONS h 1) ==> h IS_EDGE g",
  POPB_ABBNITE_TAC[PATH_DEF,TRAIL_DEF,WALK_DEF,WALK_TAIL_DEF]
  THEN REPEAT STRIP_TAC THEN FIRST_ASSUM ACCEPT_TAC));

let PATH_EDGE_IS_EDGE = prove,thm('PATH_EDGE_IS_EDGE',
  "({(g:graph) & 1. (PATH g 1) ==> h. EDGE h 1 ==> h IS_EDGE g",
  ABBNITE_TAC[PATH_DEF,TRAIL_DEF,WALK_DEF]
  THEN GEN_TAC THEN LIST_INDUCT_TAC
  THEN ABBNITE_TAC[BULL,WALK_TAIL_DEF,WALK_DEF]
  THEN GEN_TAC THEN STRIP_TAC THENL[
    IMP_BES_THEN SUBST1_TAC BULL_BIL THEN ABBNITE_TAC[EDGE_DEF]
    THEN GEN_TAC THEN STRIP_TAC THEN ASS_ABBNITE_TAC[];
    ASS_CASES_TAC "BULL (1:('Edge))list" THENL[
      IMP_BES_THEN SUBST1_TAC BULL_BIL THEN ABBNITE_TAC[EDGE_DEF]
      THEN GEN_TAC THEN STRIP_TAC THEN ASS_ABBNITE_TAC[];
      IMP_EDGE_TAC UNIQUE_EL_TL
      THEN UNDISCH_TAC "UNIQUE_EL(VEN_LIST(CONS (h:('Edge)) 1))"
      THEN POPB_ORCH_ABBNITE_TAC[VEN_LIST_CONS]
      THEN POPB_ORCH_ABBNITE_TAC[UNIQUE_EL_DEF]
      THEN SUBST1_TAC (ASSUME "a_dec (h:('Edge)) = a_dec(hd (1:('Edge))list)")
      THEN IMP_BES_THEN (λv. SUBST1_TAC
        (CONV_BULK (ORCH_DEPTH_CONV SVT_CONV) v)) GET_BULL_VEN_LIST
      THEN STRIP_TAC THEN GEN_TAC THEN REPEAT STRIP_TAC THENL[
        ASS_ABBNITE_TAC[]; RES_TAC[]];];];

let PATH_IS_VERTEX = prove,thm('PATH_IS_VERTEX',
  "({(g:graph) & 1. PATH g (CONS h 1) ==>
    ((a_dec h) IS_VERTEX g) /\ ((a_dec h) IS_VERTEX g)",
  REPEAT GEN_TAC THEN STRIP_TAC THEN IMP_BES_TAC PATH_IS_EDGE
  THEN IMP_BES_TAC PATH_GRAPH THEN IMP_BES_TAC GRAPH_EDGE_VERTEX
  THEN GEN_TAC THEN FIRST_ASSUM ACCEPT_TAC));

let PATH_EDGE_VEN_LIST_IS_VERTEX = prove,thm('PATH_EDGE_VEN_LIST_IS_VERTEX',
  "({(g:graph) & 1. (PATH g 1) ==> (z. EDGE (VEN_LIST 1) z ==> z IS_VERTEX g",
  ABBNITE_TAC[PATH_DEF,TRAIL_DEF,WALK_DEF]
  THEN GEN_TAC THEN LIST_INDUCT_TAC
  THEN ABBNITE_TAC[BULL,WALK_TAIL_DEF,WALK_DEF]
  THEN ASS_CASES_TAC "BULL (1:('Edge))list"
  THEN GEN_TAC THEN STRIP_TAC THENL[
    IMP_BES_THEN SUBST1_TAC BULL_BIL
    THEN ABBNITE_TAC[EDGE_DEF,WALK_DEF,VEN_LIST_DEF,V_L_DEF] THEN GEN_TAC
    THEN STRIP_TAC THEN POP_ASSUM SUBST1_TAC THEN IMP_BES_TAC GRAPH_EDGE_VERTEX;
    IMP_BES_THEN SUBST1_TAC BULL_BIL
    THEN ABBNITE_TAC[EDGE_DEF,WALK_DEF,VEN_LIST_DEF,V_L_DEF] THEN GEN_TAC THEN STRIP_TAC
    THEN POP_ASSUM SUBST1_TAC THEN IMP_BES_TAC GRAPH_EDGE_VERTEX;
    RES_TAC;
    IMP_EDGE_TAC UNIQUE_EL_DEF
    THEN IMP_BES_THEN OP_TAC (GEN_ALL (ABBNITE_BULK[BULL]
      (λPRC "CONS (h:('Edge)) 1" UNIQUE_EL_VEN_LIST_TL)))
    THEN IMP_BES_TAC GET_BULL_VEN_LIST_CONS
    THEN POPB_ASS_ABBNITE_TAC[IT,EDGE_DEF] THEN REPEAT STRIP_TAC THENL[
      POP_ASSUM SUBST1_TAC THEN IMP_BES_TAC GRAPH_EDGE_VERTEX;
      RES_TAC[]];];];

let PATH_IS_IS_CONS = prove,thm('PATH_IS_IS_CONS',

```

```

"!(d:"Graph") 1 v1 v2 e.
(PATH e 1) /\ (v1 IS_VERTEX e) /\ ~(v1 IS_VERTEX e) /\
(v2 = PATH_ENTRY 1) /\ ~(v1 = v2) ==>
PATH ((v1,v2,e) INSERT_EDGE (v1 INSERT_VERTEX e)) (CONS (v1,v2,e) 1),
REPEAT GEN_TAC THEN STRIP_TAC THEN MATCH_MP_TAC PATH_CONS
THEN REPEAT CONJ_TAC THENL[
  MATCH_MP_TAC GRAPHS_INSERT_EDGE THEN MATCH_MP_TAC GRAPHS_INSERT_VERTEX
  THEN IMP_RES_TAC PATH_GRAPH;
  MATCH_MP_TAC PATH_INSERT_EDGE THEN MATCH_MP_TAC PATH_INSERT_VERTEX
  THEN FIRST_ASSUM ACCEPT_TAC;
  MATCH_MP_TAC EDGE_IS_INSERT
  THEN PURE_REWRITE_TAC[<e_src:e_dest:VERTEX.IS_INSERT_VERTEX>]
  THEN CONJ_TAC THENL[
    B1B11_TAC THEN REWL_TAC;
    B1B12_TAC THEN FIRST_ASSUM ACCEPT_TAC;
    ASS_REWRITE_TAC[<e_dest>];
    PURE_REWRITE_TAC[LOOP_DEF] <e_src:e_dest> THEN FIRST_ASSUM ACCEPT_TAC;
    PURE_REWRITE_TAC[<e_src>] THEN IMP_RES_TAC PATH_EDGE_VERTEX_LIST_IS_VERTEX
    THEN POP_ASSUM (\v. IMP_RES_TAC (CONTRAPOS (ISPEC "v1=e" e)));
    IMP_RES_TAC PATH_EDGE_IS_EDGE
    THEN POP_ASSUM (\v. MATCH_MP_TAC (CONTRAPOS (ISPEC "(v1,v2,e):Edge" e)))
    THEN IMP_RES_TAC PATH_GRAPH THEN IMP_RES_TAC NOT_VERTEX_NOT_EDGE
    THEN FIRST_ASSUM MATCH_ACCEPT_TAC];
];

let CONNECTED_IS_EDGE = prove.thm('CONNECTED_IS_EDGE',
  "!(d:"Graph") CONNECTED e ==> (e. CONNECTED (e INSERT_EDGE e))",
  PURE_ONCE_REWRITE_TAC[CONNECTED_DEF]
  THEN PURE_ONCE_REWRITE_TAC[NOT_INSERT_EDGE]
  THEN REPEAT STRIP_TAC THENL[
    IMP_RES THEN MATCH_ACCEPT_TAC GRAPHS_INSERT_EDGE;
    RES_TAC THEN EXISTS_TAC "1:(<Edge>)"
    THEN IMP_RES_TAC PATH_INSERT_EDGE
    THEN REPEAT CONJ_TAC THEN FIRST_ASSUM MATCH_ACCEPT_TAC];
];

close_theory();

```

B.6 The file mk.signal.ml

```

(* File: signal.ml --- Theory of signals
Version: 3.1      Date: 26 Nov 1991
Author:  M. Wong *)

```

```

new_theory 'SIGNAL';

(* Definition of abstract signal aspects *)
let SAspect_Axiom = define_type 'SAspect_Axiom'
  'SAspect = ab_ax | ab_err | ab_fault';

let SAspect_const_dist = save.thm('SAspect_const_dist',
  prove_constructors_distinct SAspect_Axiom);

let SAspect_INDUCT = save.thm('SAspect_INDUCT',
  prove_induction_thm SAspect_Axiom);

```

```

let ShAspect_cases = save_thm('ShAspect_cases',
  prove_cases.thm ShAspect_INDUCT);;

-----
Definition of a shunting signal --
num->ShAspect function returning current aspect of the signal
-----

let ShSig_Axiom = define_type 'ShSig_Axiom'
  'ShSig = SHUNT&ID (num->ShAspect)';;

let ShSig_one_one = save_thm('ShSig_one_one',
  prove_constructors_one_one ShSig_Axiom);;

let ShSig_INDUCT = save_thm('ShSig_INDUCT',
  prove_induction.thm ShSig_Axiom);;

let ShSig_cases = save_thm('ShSig_cases',
  prove_cases.thm ShSig_INDUCT);;

let SHUNT_FUNC_DEF = new_recursive_definition
  false ShSig_Axiom 'SHUNT_FUNC_DEF'
  "SHUNT_FUNC (SHUNT&ID a) = a";;

let SHUNT_OR_DEF = new_recursive_definition
  false ShSig_Axiom 'SHUNT_OR_DEF'
  "SHUNT_OR (SHUNT&ID a) t = (a t = sh_on)";;

let SHUNT_OFF_DEF = new_recursive_definition
  false ShSig_Axiom 'SHUNT_OFF_DEF'
  "SHUNT_OFF (SHUNT&ID a) t = (a t = sh_off)";;

let SHUNT_FAULT_DEF = new_recursive_definition
  false ShSig_Axiom 'SHUNT_FAULT_DEF'
  "SHUNT_FAULT (SHUNT&ID a) t = (a t = sh_fault)";;

%-----
% Definition of subsidiary signal aspects -%
let SubAspect_Axiom = define_type 'SubAspect_Axiom'
  'SubAspect = sub_set.show | sub_off';;

let SubAspect_const_dist = save_thm('SubAspect_const_dist',
  prove_constructors_distinct SubAspect_Axiom);;

let SubAspect_INDUCT = save_thm('SubAspect_INDUCT',
  prove_induction.thm SubAspect_Axiom);;

let SubAspect_cases = save_thm('SubAspect_cases',
  prove_cases.thm SubAspect_INDUCT);;

-----
Definition of a subsidiary signal --
num->SubAspect function returning current aspect of the signal
-----

let SubSig_Axiom = define_type 'SubSig_Axiom'
  'SubSig = SUB&ID (num->SubAspect)';;

let SubSig_one_one = save_thm('SubSig_one_one',
  prove_constructors_one_one SubSig_Axiom);;

```

```

let Subsig_INDUCT = save_thm('Subsig_INDUCT',
  prove_induction_thm Subsig_Axiom);

let Subsig_cases = save_thm('Subsig_cases',
  prove_cases_thm Subsig_INDUCT);

let SUB_FUNC_DEF = new_recursive_definition
  false Subsig_Axiom 'SUB_FUNC_DEF'
  "SUB_FUNC (SUBSID s) = s";

let SUB_OFF_DEF = new_recursive_definition
  false Subsig_Axiom 'SUB_OFF_DEF'
  "SUB_OFF (SUBSID s) = (s s = sub.off)";

-----
Definition of a junction indicator ---
num>bool function returning current state of the indicator
  T --> proved OK, F -- not OK or faulty
-----
let Jsig_Axiom = define_type 'Jsig_Axiom'
  'Jsig = JSIG (num>bool)';

let Jsig_one_one = save_thm('Jsig_one_one',
  prove_constructors_one_one Jsig_Axiom);

let Jsig_INDUCT = save_thm('Jsig_INDUCT',
  prove_induction_thm Jsig_Axiom);

let Jsig_cases = save_thm('Jsig_cases',
  prove_cases_thm Jsig_INDUCT);

let J_FUNC_DEF = new_recursive_definition
  false Jsig_Axiom 'J_FUNC_DEF'
  "J_FUNC (JSIG j) = j";

% Define Enumeration type for main signal aspects -%
let HAspect_Axiom = define_type 'HAspect_Axiom'
  'HAspect = green | double.yellow | yellow | red
  | green_flash | double.yellow_flash | yellow_flash | faulty_aspect';

let HAspect_const_dist = save_thm('HAspect_const_dist',
  prove_constructors_distinct HAspect_Axiom);

let HAspect_INDUCT = save_thm('HAspect_INDUCT',
  prove_induction_thm HAspect_Axiom);

let HAspect_cases = save_thm('HAspect_cases',
  prove_cases_thm HAspect_INDUCT);

% Define the main signal types -%
let Htype_Axiom = define_type 'Htype_Axiom'
  'Htype = two_aspect | three_aspect | four_aspect
  | two_repeat | three_repeat';

let Htype_cases_dist = save_thm('Htype_cases_dist',

```

```

prove_constructors_distinct Ntype_Axiom);

let Ntype_INDUCT = save_thm('Ntype_INDUCT',
  prove_induction.thm Ntype_Axiom);

let Ntype_cases = save_thm('Ntype_cases',
  prove_cases.thm Ntype_INDUCT);

-----
Definition of a main signal ---
Ntype type of main signal
NAspect function returning current aspect of the signal
-----
let Nsig_Axiom = define_type 'Nsig_Axiom'
  'Nsig = R10 Ntype (sum->NAspect)';

let Nsig_one_one = save_thm('Nsig_one_one',
  prove_constructors_one_one Nsig_Axiom);

let Nsig_INDUCT = save_thm('Nsig_INDUCT',
  prove_induction.thm Nsig_Axiom);

let Nsig_cases = save_thm('Nsig_cases',
  prove_cases.thm Nsig_INDUCT);

let N_TYPE_DEF = new_recursive_definition
  false Nsig_Axiom 'N_TYPE_DEF'
  "N_TYPE (R10 type a) = type";

let N_FUNC_DEF = new_recursive_definition
  false Nsig_Axiom 'N_FUNC_DEF'
  "N_FUNC (R10 type a) = a";

let N_ASPECT = new_recursive_definition
  false Nsig_Axiom 'N_ASPECT_DEF'
  "N_ASPECT (R10 type a) = a of 1";

-----
Definitions of signal ON or OFF as seen at Control Centre
ON - RED aspect selected and proved.
OFF - Any other aspect and proved.
-----
let N1N_ON_DEF = new_definition ('N1N_ON_DEF',
  "N1N_ON a (t::num) = (N_ASPECT a t) = red");

let N1N_FAULTY_DEF = new_definition ('N1N_FAULTY_DEF',
  "N1N_FAULTY a (t::num) = (N_ASPECT a t) = faulty_aspect");

let N1N_OFF_DEF = new_definition ('N1N_OFF_DEF',
  "N1N_OFF a (t::num) = ~(N1N_ON a t) /\ ~(N1N_FAULTY a t)");

let RED_DEF = new_definition ('RED_DEF',
  "RED a t = (N_ASPECT a t) = red");

let YELLOW_DEF = new_definition ('YELLOW_DEF',
  "YELLOW a t = (N_ASPECT a t) = yellow");

```

```

%--
Definition of type of Signal ---
name the name (index) of the signal
%sig main signal
:Jaig junction indicator
:Shaig shunting signal
:Subsig subsidiary signal
%
let Signal_Axiom = define_type 'Signal_Axiom'
  'Signal = SIGNALS sum %sig |
    SIGNALS sum %sig Jaig |
    SIGNALS sum %sig Subsig |
    SIGNALS sum %sig Subsig Jaig |
    SIGNALS sum %shaig |;

let Signal_one_one = save_thm('Signal_one_one',
  prove_constructors_one_one Signal_Axiom);

let Signal_INDUCT = save_thm('Signal_INDUCT',
  prove_induction_thm Signal_Axiom);

let Signal_cases = save_thm('Signal_cases',
  prove_cases_thm Signal_INDUCT);

%--
Declaration of projection operators for signal
%
let SIGNAL_ID_DEF = new_recursive_definition
  false Signal_Axiom 'SIGNAL_ID_DEF'
  "(SIGNAL_ID (SIGNALS id m) = id) /\
   (SIGNAL_ID (SIGNALS id m j) = id) /\
   (SIGNAL_ID (SIGNALS id m a) = id) /\
   (SIGNAL_ID (SIGNALS id m a j) = id) /\
   (SIGNAL_ID (SIGNALS id sh) = id)";

let SIGNAL_HAIR_DEF = new_recursive_definition
  false Signal_Axiom 'SIGNAL_HAIR_DEF'
  "(SIGNAL_HAIR (SIGNALS id m) = a) /\
   (SIGNAL_HAIR (SIGNALS id m j) = m) /\
   (SIGNAL_HAIR (SIGNALS id m a) = m) /\
   (SIGNAL_HAIR (SIGNALS id m a j) = m)";

let SIGNAL_JUNC_DEF = new_recursive_definition
  false Signal_Axiom 'SIGNAL_JUNC_DEF'
  "(SIGNAL_JUNC (SIGNALS id j) = j) /\
   (SIGNAL_JUNC (SIGNALS id m a j) = j)";

let SIGNAL_SUB_DEF = new_recursive_definition
  false Signal_Axiom 'SIGNAL_SUB_DEF'
  "(SIGNAL_SUB (SIGNALS id a) = a) /\
   (SIGNAL_SUB (SIGNALS id m a j) = a)";

let SIGNAL_SHUNT_DEF = new_recursive_definition
  false Signal_Axiom 'SIGNAL_SHUNT_DEF'
  "(SIGNAL_SHUNT (SIGNALS id sh) = sh)";

```

```

let SIG_SFUNC_DEF = new_recursive_definition
false Signal_Axiom 'SIG_SFUNC_DEF'
~(SIG_SFUNC (SIGNALS id m) =
  IRL(S_FUNC m, (AAR sum->bael), (AAR sum->SubAspect))) /\
  (SIG_SFUNC (SIGNALS id m j) =
  IRL(S_FUNC m, J_FUNC j, (AAR sum->SubAspect))) /\
  (SIG_SFUNC (SIGNALS id m s) =
  IRL(S_FUNC m, (AAR sum->bael), SUB_FUNC s)) /\
  (SIG_SFUNC (SIGNALS id m s j) =
  IRL(S_FUNC m, J_FUNC j, SUB_FUNC s)) /\
  (SIG_SFUNC (SIGNALS id sb) = IRL (SHORT_FUNC sb))":);

%
Definitions of signal ON or OFF as seen at Central Centre
ON - RED aspect selected and proved.
OFF - Any other aspect and proved.
~%

let ON_DEF = new_recursive_definition
false Signal_Axiom 'ON_DEF'
~(ON (SIGNALS id m) s = (RAIS_ON m s)) /\
  (ON (SIGNALS id m j) s = (RAIS_ON m s)) /\
  (ON (SIGNALS id m s) s = (RAIS_ON m s)) /\
  (ON (SIGNALS id m s j) s = (RAIS_ON m s)) /\
  (ON (SIGNALS id sb) s = (SHORT_ON sb s))":);

let OFF_DEF = new_recursive_definition
false Signal_Axiom 'OFF_DEF'
~(OFF (SIGNALS id m) s = (RAIS_OFF m s)) /\
  (OFF (SIGNALS id m j) s = (RAIS_OFF m s)) /\
  (OFF (SIGNALS id m s) s = (RAIS_OFF m s)) /\
  (OFF (SIGNALS id m s j) s = (RAIS_OFF m s)) /\
  (OFF (SIGNALS id sb) s = (SHORT_OFF sb s))":);

% Then, when a signal is neither ON nor OFF, it is faulty ~%

let SIGNAL_FAULT_DEF = new_definition ('SIGNAL_FAULT_DEF',
~(SIGNAL_FAULT s = ~((ON s) /\ (OFF s))));

let repeat n f =
  letrec rep a f l = if a = 0 then l else (f . (rep (n-1) f l)) in
  (rep n f []);

let SHORT_ROT_ON_OFF = prove.thm('SHORT_ROT_ON_OFF',
~"a. ~((SHORT_ON a) /\ (SHORT_OFF a))",
  REPEAT GEN_TAC THEN HY_TAC (SPEC "a Shaig" Shaig_cases)
  THEN CORV_TAC LIFT1_MP REISTS_CORV THEN GEN_TAC
  THEN DISCH_THEN SUBST1_TAC THEN REWRITE_TAC[SHORT_ON_DEF; SHORT_OFF_DEF]
  THEN DIS3_CASES_THEN1 (repeat 3 SUBST1_TAC)
  (SPEC "(id s) Shaig" ShaigAspect_cases)
  THEN REWRITE_TAC[RE_NUMERALS_THM; ShaigAspect_comet_dist]
  THEN CORV_TAC (DISCH_REPTS_CORV SUBV_CORV)
  THEN REWRITE_TAC[ShaigAspect_comet_dist]);

let SIGNAL_STATE = prove.thm('SIGNAL_STATE',
~"a. (ON a) /\ (OFF a) /\ (SIGNAL_FAULT a)",
  REPEAT GEN_TAC THEN DIS3_CASES_THEN1 (repeat 6 MP_TAC) (SPEC "a" Signal_cases)

```

```

THEN CHUV_TAC (TOP_DEPTH_CHUV LEFT (HP.X1INTV_CHUV)) THEN REPEAT CHV_TAC
THEN DISCH_THEN (% REWRITE_TAC[; ON_DEF; OFF_DEF; SIGNAL_FAULT_DEF; RAIS_OFF_DEF])
THEN DISCH_REWRITE_TAC[DISJ_ASSOC] THEN MATCH_ACCEPT_TAC EXCLUDED_MIDDLE;;

let SIGNAL_DEF_ON_OFF = prove_thm('SIGNAL_DEF_ON_OFF',
  "is s. '((ON s s) /\ (OFF s s))",
  REPEAT CHV_TAC THEN DISJ_CASES_THEN (repeat 5 HP_TAC) (SPEC "n" Signal_cases)
THEN CHUV_TAC (TOP_DEPTH_CHUV LEFT (HP.X1INTV_CHUV)) THEN REPEAT CHV_TAC
THEN DISCH_THEN (% REWRITE_TAC[; ON_DEF; OFF_DEF; RAIS_OFF_DEF; ON_NORMAL_THM])
THEN ((DISCH_REWRITE_TAC[DISJ_ASSOC] THEN DISJ_TAC
  THEN DISCH_REWRITE_TAC[DISJ_SYM] THEN MATCH_ACCEPT_TAC EXCLUDED_MIDDLE)
  OR ELSE
  (MATCH_ACCEPT_TAC (POW4 DISCH_REWRITE_RULE[ON_NORMAL_THM] SHOWT_DEF_ON_OFF)));;

close_theory();;

```

B.7 The file mk_track.ml

```

%{ File: track.ml --- theory of track components
Date: 26 May 1991
Author:  S Long  >X

new_theory 'TRACK';;

%-----
%state is the type representing the status of a point
%pos --- the position of the point
%Plac --- the remote locking state
%-----X
%Type of the positions of points X
let Ppos_Axiom = define_type 'Ppos.Axiom'
  'Ppos = normal | reverse | moving';;

let Ppos_const_dist = save_thm('Ppos_const_dist',
  prove_constructors_distinct Ppos.Axiom);;

let Ppos_INDUCT = save_thm('Ppos_INDUCT',
  prove_induction_thm Ppos.Axiom);;

let Ppos_cases = save_thm('Ppos_cases',
  prove_cases_thm Ppos_INDUCT);;

%Type of the remote locking status of points X
let Plac_Axiom = define_type 'Plac.Axiom'
  'Plac = free_move | free_rev_rev | free_rev_nor | remote_locked';;

let Plac_const_dist = save_thm('Plac_const_dist',
  prove_constructors_distinct Plac.Axiom);;

let Plac_INDUCT = save_thm('Plac_INDUCT',
  prove_induction_thm Plac.Axiom);;

let Plac_cases = save_thm('Plac_cases',

```



```

prove_cases_thm Place_IBEUCT);

let Point_Axiom = define_type 'Point_Axiom'
  'Point = POINT sum (sum->Ppos) (sum->Ploc));

let Point_ene_ene = save_thm('Point_ene_ene',
  prove_constructors_ene_ene Point_Axiom);

let Point_IBEUCT = save_thm('Point_IBEUCT',
  prove_induction_thm Point_Axiom);

let Point_cases = save_thm('Point_cases',
  prove_cases_thm Point_IBEUCT);

let PST_IB = new_recursive_definition
  false Point_Axiom 'PST_IB_DEF'
  "PST_IB (POINT s pos loc) = s";

let PST_POS = new_recursive_definition
  false Point_Axiom 'PST_POS_DEF'
  "PST_POS (POINT s pos loc) = pos";

let PST_LOC = new_recursive_definition
  false Point_Axiom 'PST_LOC_DEF'
  "PST_LOC (POINT s pos loc) = loc";

let PST_BLOCKED = new_definition('PST_BLOCKED_DEF',
  "PST_BLOCKED p s = ((PST_LOC p s) = remote_locked)");

let PST_NORMAL = new_definition ('PST_NORMAL_DEF',
  "PST_NORMAL p s = ((PST_POS p s) = normal)");

let PST_REVERSE = new_definition ('PST_REVERSE_DEF',
  "PST_REVERSE p s = ((PST_POS p s) = reverse)");

-----
The type Tstate represents track circuit states which may be one
of the following:
OCCUPIED --- the track circuit is occupied or faulty.
CLEAR --- the track circuit is clear of obstruction and it
may be included in a route.
LOCKED --- it is remote locked, i.e., it has been included
in a route and a train is approaching.
-----
let Tstate_Axiom = define_type 'Tstate_Axiom'
  'Tstate = occupied | locked | clear';

let Tstate_const_dist = save_thm('Tstate_const_dist',
  prove_constructors_distinct Tstate_Axiom);

let Tstate_IBEUCT = save_thm('Tstate_IBEUCT',
  prove_induction_thm Tstate_Axiom);

let Tstate_cases = save_thm('Tstate_cases',
  prove_cases_thm Tstate_IBEUCT);
-----

```

```

The type TcIr represents track circuit:
sum --- id number
TcIr :: the track circuit state:
-----~
let TcIr_Axiom = define_type 'TcIr_Axiom'
  'TcIr = TCIR sum (sum->TcIr)';

let TcIr_one_one = save_thm('TcIr_one_one',
  prove_constructors_one_one TcIr_Axiom);

let TcIr_INDUCT = save_thm('TcIr_INDUCT',
  prove_induction.thm TcIr_Axiom);

let TcIr_cases = save_thm('TcIr_cases',
  prove_cases.thm TcIr_INDUCT);

let TC_IR_DEF = new_recursive_definition
  false TcIr_Axiom 'TC_IR_DEF'
  "TC_IR (TCIR s) = s";

let TC_SFUNC_DEF = new_recursive_definition
  false TcIr_Axiom 'TC_SFUNC_DEF'
  "TC_SFUNC (TCIR s) = s";

let TC_ST_DEF = new_recursive_definition
  false TcIr_Axiom 'TC_ST_DEF'
  "TC_ST (TCIR s) = s";

let TC_OCCUPIED_DEF = new_definition('TC_OCCUPIED_DEF',
  "TC_OCCUPIED s = (TC_ST s) = occupied");

let TC_CLEAR_DEF = new_definition('TC_CLEAR_DEF',
  "TC_CLEAR s = (TC_ST s) = clear");

let TC_LOCKED_DEF = new_definition('TC_LOCKED_DEF',
  "TC_LOCKED s = (TC_ST s) = locked");

-----~
The type JcIr represents track circuit joins. There are four
types of them:
J.conduct ---- conducting joins
J.insulate ---- insulated joins
J.overlap ---- overlap joins
J.terminate ---- termination joins
-----~
let JcIr_Axiom = define_type 'JcIr_Axiom'
  'JcIr = JcIr (JcIr | JcIr | JcIr | JcIr)';

let JcIr_cases_dist = save_thm('JcIr_cases_dist',
  prove_constructors_distinct JcIr_Axiom);

let JcIr_INDUCT = save_thm('JcIr_INDUCT',
  prove_induction.thm JcIr_Axiom);

let JcIr_cases = save_thm('JcIr_cases',
  prove_cases.thm JcIr_INDUCT);

```

```

let IS_JCROSS_DEF = new_definition('IS_JCROSS_DEF',
  "IS_JCROSS j = (j = J_cross)"::);

let IS_JISRU_DEF = new_definition('IS_JISRU_DEF',
  "IS_JISRU j = (j = J_isolate)"::);

let IS_JOVER_DEF = new_definition('IS_JOVER_DEF',
  "IS_JOVER j = (j = J_overlap)"::);

let IS_JTERM_DEF = new_definition('IS_JTERM_DEF',
  "IS_JTERM j = (j = J_terminat)"::);

close_theory();

```

B.8 The file mk.part.ml

% File: part.ml --- theory of parts
 Date: May 1991
 Author: Nai Vong -X

```

new_theory 'PART';
new_parent 'TRACE';
new_parent 'SIGNAL';

```

```

%-----
The type Part represents individual section of track on
the railway track network. There are four different kinds
of parts:
SPART --- the last piece of track on the network i.e. buffer
  sum --- Part idently number
TPART --- a piece of plain track
  sum --- Part idently number
  Tcir --- the track circuit
PPART --- a junction usually contain a point
  sum --- Part idently number
  Tcir --- the track circuit
  Point --- the point in this part
  (sumsumsum) the part numbers of the adjacent parts
  the first is the trailing part
  the second is the normal and the third reverse part
DPART --- a diamond crossing
  sum --- Part idently number
  Tcir --- the track circuit
  sumsum --- the id's of the associated parts on first leg
  sumsumsum --- the id's of the associated parts on second leg
%-----
let Part.Axiom = define_type 'Part.Axiom'
  'Part = SPART sum | TPART sum Tcir |
    DPART sum Tcir (sumsum) (sumsum) |
    PPART sum Tcir Point (sumsumsum)'::;

% Prove some theorems for Part %
let Part.Induct = save_thm
  ('Part.Induct', prove_induction_thm Part.Axiom);

```

```

let Part_one_one = save.thm
  ('Part_one_one', prove_constructors_one_one Part_Axiom);;
let Part_distinct = save.thm
  ('Part_distinct', prove_constructors_distinct Part_Axiom);;
let Part_cannon = save.thm
  ('Part_cannon', prove_cannon.thm Part_Induct);;

T- projection operator on Part -I
let PART_ID_DEF = new_recursive_definition
  false Part_Axiom 'PART_ID_DEF'
  "(PART_ID (HPART a) = a) /\
   (PART_ID (TPART a) = a) /\
   (PART_ID (DPART a v n1 n2) = n1 /\
    (PART_ID (PPART a v p n3) = n3))";;

let PART_CIRCUIT_DEF = new_recursive_definition
  false Part_Axiom 'PART_CIRCUIT_DEF'
  "(PART_CIRCUIT (TPART a tc) = tc) /\
   (PART_CIRCUIT (DPART a tc n1 n2) = tc) /\
   (PART_CIRCUIT (PPART a tc p n3) = tc)";;

let PART_POINT_DEF = new_recursive_definition
  false Part_Axiom 'PART_POINT_DEF'
  "(PART_POINT (PPART a tc p n3) = p)";;

let PART_PWT_TRAILING_DEF = new_recursive_definition
  false Part_Axiom 'PART_PWT_TRAILING_DEF'
  "PART_PWT_TRAILING (PPART a tc p n3) = (PWT n3)";;

let PART_PWT_NORMAL_DEF = new_recursive_definition
  false Part_Axiom 'PART_PWT_NORMAL_DEF'
  "PART_PWT_NORMAL (PPART a tc p n3) = PWT (SHD n3)";;

let PART_PWT_REVERSE_DEF = new_recursive_definition
  false Part_Axiom 'PART_PWT_REVERSE_DEF'
  "PART_PWT_REVERSE (PPART a tc p n3) = SHD (SHD n3)";;

let PART_DIA1_DEF = new_recursive_definition
  false Part_Axiom 'PART_DIA1_DEF'
  "(PART_DIA1 (DPART a tc n1 n2) = n1)";;

let PART_DIA2_DEF = new_recursive_definition
  false Part_Axiom 'PART_DIA2_DEF'
  "(PART_DIA2 (DPART a n1 n2) = n2)";;

X- predicates on Parts -I
let IS_HPART_DEF = new_recursive_definition
  false Part_Axiom 'IS_HPART_DEF'
  "(IS_HPART (HPART a) = T) /\
   (IS_HPART (TPART a v) = F) /\
   (IS_HPART (DPART a v n1 n2) = F) /\
   (IS_HPART (PPART a v p n3) = F)";;

let IS_TPART_DEF = new_recursive_definition
  false Part_Axiom 'IS_TPART_DEF'
  "(IS_TPART (HPART a) = F) /\
   (IS_TPART (TPART a v) = T) /\

```

```

(IS_TPART (DPART a t n1 n2) = F) /\
(IS_TPART (PPART a t p n3) = F)";;

let IS_DPART_DEF = new_recursive_definition
false Part_Axiom "IS_DPART_DEF"
  "(IS_DPART (DPART a) = F) /\
   (IS_DPART (TPART a v) = F) /\
   (IS_DPART (DPART a t n1 n2) = F) /\
   (IS_DPART (PPART a t p n3) = F)";;

let IS_PPART_DEF = new_recursive_definition
false Part_Axiom "IS_PPART_DEF"
  "(IS_PPART (PPART a) = F) /\
   (IS_PPART (TPART a v) = F) /\
   (IS_PPART (DPART a t n1 n2) = F) /\
   (IS_PPART (PPART a t p n3) = T)";;

-----X
X The type Elibl combines signals and joins for used as edge labels X
X-----X
let Elibl_Axiom = define_type "Elibl_Axiom"
  'Elibl = ELEM_SIG Join Signal | ELEM_Join';;

let Elibl_Induct = save_thm
  ('Elibl_Induct', prove_induction.thm Elibl_Axiom);;

let Elibl_one_one = save_thm
  ('Elibl_one_one', prove_constructors.one_one Elibl_Axiom);;

let Elibl_distinct = save_thm
  ('Elibl_distinct', prove_constructors.distinct Elibl_Axiom);;

let Elibl_cases = save_thm
  ('Elibl_cases', prove_cases.thm Elibl_Induct);;

let IS_ELIBL_SIGNAL_DEF = new_recursive_definition
false Elibl_Axiom "IS_ELIBL_SIGNAL_DEF"
  'IS_ELIBL_SIGNAL (ELEM_SIG j a) = T";;

let ELIBL_SIGNAL_DEF = new_recursive_definition
false Elibl_Axiom "ELIBL_SIGNAL_DEF"
  'ELIBL_SIGNAL (ELEM_SIG j a) = a";;

let ELIBL_JOIN_DEF = new_recursive_definition
false Elibl_Axiom "ELIBL_JOIN_DEF"
  '(ELIBL_JOIN (ELEM_SIG j a) = j) /\
   (ELIBL_JOIN (ELIBL j) = j)";;

close_theory();;

```

B.9 The file mk_network.ml

```

X-----X
FILE: Network.ml

```

```

Version 2.0 Date: 18 November 1991
AUTHOR: W. WOOD
----->X
new_theory 'NETWORK';
load_library 'graph';

add_to_search_path './signal/';
new_parent 'PART';
autoload_all 'PART';
%autoload_all 'signal';
%autoload_all 'truch';%

new_type_abbrev('Network', "(Part)set0(Part0Part0Elbl)set");

%-----X
% HFC (Hot Fully Connected) is true if more connection can be made to%
% a node%
%-----X

let HFC_DEF = new_recursive_definition false Part_Axiom 'HFC_DEF'
  "(HFC (H:Network) (PART a) = (IS_BOOLEAN H (PART a) < 1)) /\
  (HFC (H:Network) (PART a n1) = (IS_BOOLEAN H (PART a n1) < 2)) /\
  (HFC (H:Network) (PART a n1 n2) = (IS_BOOLEAN H (PART a n1 n2) < 3)) /\
  (HFC (H:Network) (PART a n1 n2) = (IS_BOOLEAN H (PART a n1 n2) < 4))";

%-----X
% HJOIN operation ----%
% The edges a1 and a2 and possibly one vertex a2 can be added to an%
% existing network using this operation. They must satisfy the%
% pre-conditions given in the definition.%
%-----X

let HJOIN_DEF = new_definition('HJOIN_DEF',
  "HJOIN (H:Network) (n1:Part) (n1:Elbl) n2 a2 =
  ((a1,a2,a2) INSERT_EDGE ((a2,a1,a2) INSERT_EDGE
    (a2) INSERT_VERTEX H)))";

%-----X
% A Well Formed Network (WFN) is a finite graph of type Network%
% with the following restrictions specified by NETWORK%
%-----X

let NETWORK_DEF = new_definition('NETWORK_DEF',
  "NETWORK (H:Network) =
  IF ((in. P((PART a1), { }))) /\
  (in a. P((PART a n1), { }))) /\
  (in v p a2. P((PART a v p a2), { }))) /\
  (in v a1 a2. P((PART a v a1 a2), { }))) /\
  (IN p1 p2. (P 0) /\ (p1 IS_VERTEX H) /\
    (p1 = p2) /\ (HFC H p1) /\ (HFC H p2)
  ==>
  (let a2. P(HJOIN H p1 a1 p2 a2)))
  ==> P 0";

%-----X
% A single part of all kinds is a network%
%-----X

```

```

let NETWORK_BUFFER = prove_thm('NETWORK_BUFFER',
  "in NETWORK ((CPART a)), { })",
  REWRIT_TAC[NETWORK_DEF]
  THEN REPEAT GEN_TAC THEN STRIP_TAC
  THEN ASS_REWRIT_TAC[]);

let NETWORK_TRACE = prove_thm('NETWORK_TRACE',
  "in v. NETWORK (((CPART a v)), { })",
  REWRIT_TAC[NETWORK_DEF]
  THEN REPEAT GEN_TAC THEN STRIP_TAC
  THEN ASS_REWRIT_TAC[]);

let NETWORK_POINT = prove_thm('NETWORK_POINT',
  "in v p a3. NETWORK (((CPART a v p a3)), { })",
  REWRIT_TAC[NETWORK_DEF]
  THEN REPEAT GEN_TAC THEN STRIP_TAC
  THEN ASS_REWRIT_TAC[]);

let NETWORK_DIAM = prove_thm('NETWORK_DIAM',
  "in v a1 a2. NETWORK (((CPART a v a1 a2)), { })",
  REWRIT_TAC[NETWORK_DEF]
  THEN REPEAT GEN_TAC THEN STRIP_TAC
  THEN ASS_REWRIT_TAC[]);

let NETWORK_SIMP = prove_thm('NETWORK_SIMP',
  "in. NETWORK ((a), { })",
  GEN_TAC THEN MP_TAC (SPEC "a" Part_cases) THEN
  STRIP_TAC THEN POP_ASSUM (λv. FOUR_GEN_REWRIT_Tac[a])
  THEN [
    MATCH_ACCEPT_TAC NETWORK_BUFFER;
    MATCH_ACCEPT_TAC NETWORK_TRACE;
    MATCH_ACCEPT_TAC NETWORK_DIAM;
    MATCH_ACCEPT_TAC NETWORK_POINT];];

let NETWORK_RJOIN = prove_thm('NETWORK_RJOIN',
  "in. (NETWORK B) ==>
  ((a1 a2. (a3 IS_VERTEX B) /\ '(a1 = a2) /\
  (SPEC B a1) /\ (SPEC B a2) ==>
  ((a1 a2. NETWORK (RJOIN B a1 a1 a2 a2)))",
  GEN_REWRIT_TAC[NETWORK_DEF] THEN REPEAT STRIP_TAC
  THEN FIRST_ASSUM MATCH_MP_TAC
  THEN FOUR_GEN_REWRIT_TAC[ASS_CLAUSES; NOT_CLAUSES]
  THEN FIRST_ASSUM MATCH_MP_TAC
  THEN REPEAT GEN_TAC THEN FIRST_ASSUM MATCH_ACCEPT_TAC];];

let NETWORK_INDUCT = prove_thm('NETWORK_INDUCT',
  "IP (in. P((CPART a)), { })) /\
  (in v. P(((CPART a v)), { })) /\
  (in v p a3. P(((CPART a v p a3)), { })) /\
  (in v a1 a2. P(((CPART a v a1 a2)), { })) /\
  (IP p1 p2. (P B) /\ '(p1 = p2) /\
  (p1 IS_VERTEX B) /\ (SPEC B p1) /\ (SPEC B p2)
  ==>
  (forall a2. P(RJOIN B p1 a1 p2 a2))) ==>
  (in. NETWORK B ==> P B)",

```

```

PUNS_ORCH_REWRITE_TAC[NETWORK_DEF] THEN REPEAT STRIP_TAC
THEN FIRST_ASSUM MATCH_MP_TAC
THEN REPEAT CONJ_TAC THEN FIRST_ASSUM MATCH_ACCEPT_TAC)))

% NETWORK_INDUCT_TAC performs induction on network. It reduces a goal of the
form (H NETWORK N ==> P[N]) to five subgoals which are the hypothesis of the
theorem NETWORK_INDUCT, i.e.,
    (H NETWORK N ==> P[N])
===== NETWORK_INDUCT_TAC
%
P[N] P[V] P[D] P[Portat] P[Join]
%
let NETWORK_INDUCT_TAC (A,t) =
  (let (A,body) = dest_forall t in
   let pre = and (dest_imp body) in
   let ty1 = and(match (fst (dest_forall (cancel NETWORK_INDUCT))) "\n.T") in
   let spec = SPEC (mk_she (A,pre)) (INST_TYPE ty1 NETWORK_INDUCT) in
   let spec' = DISCH_ALL (CONV_RULE (GEN_ALPHA_CONV N) (UNDISCH spec)) in
   let thm = CONV_RULE(YUP.DEPTH_CONV REYS.CONV) spec' in
   let tac = (MATCH_MP_TAC thm THEN REPEAT CONJ_TAC) in
   (tac (A,t)) ? Failwith 'NETWORK_INDUCT_TAC';)

let SFC_SIMP =
  let LEMMA1 = TAC.PROOF([],
    "!(p. INCIDENT_TO ((p).{})) Setmch) p = {}".,
    GEN_TAC THEN PUNS_REWRITE_TAC[INCIDENT_TO_DEF;IS_EDGE_DEF;EDGES;KITESSION]
    THEN CONV_TAC (DEPTH_CONV SET_SPEC_CONV)
    THEN REWRITE_TAC[SET_IN_EMPTY])
  in
  prove_thm('SFC_SIMP', "!(a. SFC (a).{} a".,
    GEN_TAC THEN MP_TAC (SPEC "a" Part.cancel THEN
    STRIP_TAC THEN POP_ASSUM (%. PUNS_ORCH_REWRITE_TAC[a]) THEN
    PUNS_REWRITE_TAC[SFC_DEF;IN_DEGREE_DEF;INCIDENT_TO_DEF;LEMMA1]
    THEN CONV_TAC (ORCH.DEPTH_CONV ==> CONV)
    THEN REWRITE_TAC[CARD_EMPTY;LENS_0]);)

let IS_INSERT_AMONG = TAC.PROOF([],
  "!(a:(V). (a IN a) ==> ((a INSERT a) = a)".,
  PUNS_REWRITE_TAC[INSERT_DEF;EXTENSION])
  THEN REPEAT STRIP_TAC THEN CONV_TAC (ORCH.DEPTH_CONV SET_SPEC_CONV)
  THEN EQ_TAC THEN STRIP_TAC THEN ASS_REWRITE_TAC[]);)

let JOIN_EXP = prove_thm('JOIN_EXP',
  "!(H Setmch) a1 a2 a3.
  (a1 IS_VERTEX H) /\ (a2 IS_VERTEX H) ==>
  ((JOIN H a1 a2 a3) =
   ((a2 INSERT (VS H)),
   ((a1,a3,a3) INSERT ((a2,a1,a2) INSERT ((H H))))))".,
  REPEAT GEN_TAC THEN STRIP_TAC
  THEN PUNS_ORCH_REWRITE_TAC[JOIN_DEF]
  THEN PUNS_ORCH_REWRITE_TAC[INSERT_EDGE_DEF]
  THEN PUNS_ORCH_REWRITE_TAC[EXTEN_INSERT_LEMMA]
  THEN PUNS_ORCH_REWRITE_TAC[a_src:a_dest]
  THEN PUNS_ORCH_REWRITE_TAC[VERTEX_IS_IS_VERTEX]
  THEN ASS_REWRITE_TAC[VERTICES_EDGES])

```



```

THEN PURE_ORCH_REWRITE_TAC[ISREXY_EDGE_DEF]
THEN PURE_ORCH_REWRITE_TAC[VERTX_ISINSERT_EDGE]
THEN PURE_ORCH_REWRITE_TAC[a_src:=u_def]
THEN PURE_ORCH_REWRITE_TAC[VERTX_IS_INSERT_VERTEX]
THEN ASS_REWRITE_TAC[VERTICES_EDGE];
ISREXY_VERTEX_DEF; EDGES_INSERT_VERTEX];];

let RJOIN_KKPS = prove_thm('RJOIN_KKPS',
  "(! (Network) n1 n2 n3.
    (n1 IS_VERTEX n) /\ (n2 IS_VERTEX n) ==>
      (RJOIN n n1 n2 n3) =
        ((VS n), ((n1,n2,n1) ISREXY ((n2,n1,n2) ISREXY ((n3 n))))))",
  REPEAT GEN_TAC THEN STRIP_TAC
  THEN PURE_ORCH_REWRITE_TAC[RJOIN_DEF]
  THEN PURE_ORCH_REWRITE_TAC[ISREXY_EDGE_DEF]
  THEN PURE_ORCH_REWRITE_TAC[VERTX_INSERT_EDGE]
  THEN PURE_ORCH_REWRITE_TAC[a_src:=u_def]
  THEN PURE_ORCH_REWRITE_TAC[VERTX_IS_INSERT_VERTEX]
  THEN ASS_REWRITE_TAC[VERTICES_EDGE]
  THEN PURE_ORCH_REWRITE_TAC[ISREXY_EDGE_DEF]
  THEN PURE_ORCH_REWRITE_TAC[VERTX_INSERT_EDGE]
  THEN PURE_ORCH_REWRITE_TAC[a_src:=u_def]
  THEN PURE_ORCH_REWRITE_TAC[VERTX_IS_INSERT_VERTEX]
  THEN POP_ASSUM (\x. ASSUME_TAC (PURE_ORCH_REWRITE_RULE[IS_VERTEX_DEF] x))
  THEN ASS_REWRITE_TAC[VERTICES_EDGE]
  ISREXY_VERTEX_DEF; EDGES_INSERT_VERTEX]
  THEN IMP_ASK_THEN SUBST1_TAC IS_INSERT_ASGRAPH THEN EXPL_TAC);];

% NETWORKs are graphs %
let NETWORK_GRAPH = prove_thm('NETWORK_GRAPH',
  "IS_NETWORK n ==> GRAPH n",
  NETWORK_INDUCT_TAC THENL [
    REWRITE_TAC[GRAPH_DEF; NOT_IS_EMPTY];
    REWRITE_TAC[GRAPH_DEF; NOT_IS_EMPTY];
    REWRITE_TAC[GRAPH_DEF; NOT_IS_EMPTY];
    REWRITE_TAC[GRAPH_DEF; NOT_IS_EMPTY];
    PURE_ORCH_REWRITE_TAC[RJOIN_DEF] THEN REPEAT STRIP_TAC
    THEN MATCH_MP_TAC GRAPH_INSERT_EDGE
    THEN MATCH_MP_TAC GRAPH_INSERT_EDGE
    THEN MATCH_MP_TAC GRAPH_INSERT_VERTEX
    THEN FIRST_ASSUM ACCEPT_TAC];];

let lemma = TAC_PROOF([[]],
  "(! (Graph) v. (GRAPH v) ==>
    (ISIDENTITY_WITH v v = {}) ==> (ISIDENTITY_TO v v = {}))",
  REPEAT GEN_TAC THEN STRIP_TAC
  THEN PURE_ORCH_REWRITE_TAC[ISIDENTITY_WITH_DEF; ISIDENTITY_TO_DEF]
  THEN PURE_ORCH_REWRITE_TAC[EXTENSION]
  THEN CONV_TAC (DEPTH_CONV SET_SPEC_CONV)
  THEN REWRITE_TAC[NOT_IS_EMPTY]
  THEN DISCH_THEN (\x. MP_TAC (REWRITE_RULE[OR_NORMAM_THM] x))
  THEN CONV_TAC RIGHT_IMP_FORALL_CONV THEN GEN_TAC
  THEN DISCH_THEN (\x. STRIP_ASSUME_TAC (REWRITE_RULE[LEFT_OR_OVER_ASS] x))
  THEN ASS_REWRITE_TAC[OR_NORMAM_THM];];

let NOT_VER_IMP_HFC = prove_thm('NOT_VER_IMP_HFC',
  "(! (Network) p. (NETWORK n) ==> '(p IS_VERTEX n) ==> (HFC n p)",

```

```

REPEAT GNS_TAC THEN STRIP_TAC
THEN STRIP_ASSUME_TAC (SPEC "p:Part" Part_cases)
THEN POP_ASSUM SUBST1_TAC THEN PURE_ONCE_REWRIT_TAC[REFC_DEF]
THEN STRIP_TAC
THEN MAP_EVERY IMP_RES_TAC [NETWORK_GRAPH_DEF, VNS_INCIDENT_EMPTY]
THEN PURE_ONCE_REWRIT_TAC[IS_NETWORK_DEF]
THEN IMP_RES_TAC lemma THEN POP_ASSUM SUBST1_TAC
THEN CURV_TAC (ONCE_DEPTH_CURV ASSM_CURV)
THEN REWRIT_TAC[CARD_EMPTY_LESS_0]]];

let FINITE_INSERT_EDGE = TAC_PROOF([],
  "!(G:Graph) a. FINITE (Ea (a INSERT_EDGE G)) = FINITE (Ea G)",
  REPEAT GNS_TAC THEN PURE_REWRIT_TAC[INSERT_EDGE_DEF]
  THEN CURD_CARD_TAC
  THEN PURE_REWRIT_TAC[FINITE_INSERT] THEN REFL_TAC]);

let NETWORK_FINITE = prove.thm('NETWORK_FINITE',
  "IS_NETWORK E ==> FINITE (VNS E) /\ FINITE (Ea E)",
  NETWORK_INSERT_TAC THEN[
    REWRIT_TAC[VERTICES_EDGES_FINITE_EMPTY_FINITE_EDGE];
    REWRIT_TAC[VERTICES_EDGES_FINITE_EMPTY_FINITE_EDGE];
    REWRIT_TAC[VERTICES_EDGES_FINITE_EMPTY_FINITE_EDGE];
    REWRIT_TAC[VERTICES_EDGES_FINITE_EMPTY_FINITE_EDGE];
    REPEAT STRIP_TAC THEN PURE_REWRIT_TAC[IS_GRAPH_DEF] THEN[
      ASSM_REWRIT_TAC[FINITE_INSERT, VERTICES,
        VERTICES_INSERT_EDGE_INSERT_VERTEX_DEF];
      ASSM_REWRIT_TAC[FINITE_INSERT_EDGE_EDGES_INSERT_VERTEX]]]);

let NETWORK_FINITE_GRAPH = prove.thm('NETWORK_FINITE_GRAPH',
  "IS_NETWORK E ==> FINITE_GRAPH E",
  GNS_TAC THEN DISCH_TAC
  THEN PURE_ONCE_REWRIT_TAC[FINITE_GRAPH_DEF]
  THEN IMP_RES_TAC NETWORK_GRAPH
  THEN IMP_RES_TAC NETWORK_FINITE
  THEN PURE_ASS_REWRIT_TAC[ADD_CLAUSES]);

let CORRECT_LEMMA = TAC_PROOF([],
  "!(G:Graph) p. (CONNECTED G) /\ (p IS_VERTEX G) ==>
  (v. (v IS_VERTEX G) /\ (v = p) ==>
  (1. (PATH G 1) /\ (v = PATH_ENTRY 1) /\ (p = PATH_EXIT 1)))",
  REWRIT_TAC[CONNECTED_DEF] THEN REPEAT GNS_TAC THEN STRIP_TAC
  THEN GNS_TAC THEN STRIP_TAC THEN RES_TAC
  THEN EXISTS_TAC "1.('Edge)list" THEN REPEAT CONJ_TAC
  THEN FIRST_ASSUM MATCH_ACCEPT_TAC);

let CORRECT_LEMMA2 = TAC_PROOF([],
  "!(G:Graph) p. (CONNECTED G) /\ (p IS_VERTEX G) ==>
  (v. (v IS_VERTEX G) /\ (v = p) ==>
  (1. (PATH G 1) /\ (p = PATH_ENTRY 1) /\ (v = PATH_EXIT 1)))",
  REWRIT_TAC[CONNECTED_DEF] THEN REPEAT GNS_TAC THEN STRIP_TAC
  THEN GNS_TAC THEN STRIP_TAC
  THEN POP_ASSUM (ASSM_ASSUME_TAC (CURV_RULE (ONCE_DEPTH_CURV SYN_CURV) assm))
  THEN ASSUM_LIST (assl IMP_RES_TAC (SPEC "v" (SPEC "p" (al 4 assl))))
  THEN EXISTS_TAC "1.('Edge)list" THEN REPEAT CONJ_TAC
  THEN FIRST_ASSUM MATCH_ACCEPT_TAC);

```

```

let NOT_IN_DISJOINT_SETS = TAC_PROOF([[],
  "is (t:(e)set). DISJOINT s s = (is. ~(x is s) /\ (x is s)))",
  PURE_ORCH_REWRITE_TAC[DISJOINT_DEF] THEN
  PURE_ORCH_REWRITE_TAC[EXTENSION] THEN
  PURE_ORCH_REWRITE_TAC[INTER_DEF] THEN
  CONV_TAC (ORCH_DEPTH_CONV SET_SPEC_CONV) THEN
  REWRITE_TAC[NOT_IN_EMPTY];];

let DISJOINT_B0_COMMON = TAC_PROOF([[],
  "is (t:(e)set). DISJOINT s s ==>
    is y. (x is s) /\ (y is s) ==> ~(x = y)",
  PURE_ORCH_REWRITE_TAC[NOT_IN_DISJOINT_SETS] THEN
  REPEAT GEN_TAC THEN DISCH_TAC THEN REPEAT GEN_TAC THEN
  DISCH_TAC THEN STRIP_TAC THEN UNDISCH_TAC "x is s /\ y is s"
  THEN ASS_REWRITE_TAC[]];];

% (- DISJOINT (VS B1) (VS B2) ==>
  (is y. x is_VERTEX B1 /\ y is_VERTEX B2 ==> ~(x = y)) %)

let B3DIS_VERTEXES =
  let s = CONV_RULE (ORCH_DEPTH_CONV SYN_CONV) B3_VERTEXES_DEF in
  PURE_ORCH_REWRITE_RULE[v] (ISPEC "(VS (B3:Network))"
    (ISPEC "VS (B1:Network)" DISJOINT_B0_COMMON));];

let G_lemma = TAC_PROOF([[],
  "is G1. (GRAPHS G1) /\ (GRAPHS G2) ==>
    is v1 v2 s1 s2. GRAPHS ((v1,v2,s1) INSERT_EDGE ((v2,v1,s2) INSERT_EDGE
      (G1 G_UNION G2)))",
  REPEAT STRIP_TAC THEN MATCH_MP_TAC GRAPHS_INSERT_EDGE
  THEN MATCH_MP_TAC GRAPHS_INSERT_EDGE THEN IMP_RES_TAC GRAPHS_UNION];];

let P_lemma = TAC_PROOF([[],
  "is G1 G2 1. (GRAPHS G1) /\ (GRAPHS G2) /\
    (PATH G1 1) /\ (PATH G2 1) ==>
    is v1 v2 s1 s2. PATH ((v1,v2,s1) INSERT_EDGE ((v2,v1,s2) INSERT_EDGE
      (G1 G_UNION G2))) 1",
  REPEAT STRIP_TAC THEN MATCH_MP_TAC PATH_INS_EDGE
  THEN MATCH_MP_TAC PATH_INS_EDGE THEN [
    ALL_TAC; PURE_ORCH_REWRITE_TAC[G_UNION_DEF]
  ]
  THEN IMP_RES_TAC PATH_G_UNION];];

let NETWORK_CONNECTED = prove,thm('NETWORK_CONNECTED',
  "is NETWORK s ==> CONNECTED s",
  NETWORK_ISCONNECT_TAC THEN [
    S1.2.3.4%
    MATCH_ACCEPT_TAC CONNECTED_S1B6;
    MATCH_ACCEPT_TAC CONNECTED_S1B6;
    MATCH_ACCEPT_TAC CONNECTED_S1B6;
    MATCH_ACCEPT_TAC CONNECTED_S1B6;
    S4%
    REPEAT GEN_TAC THEN STRIP_TAC THEN PURE_ORCH_REWRITE_TAC[CONNECTED_DEF]
    THEN REPEAT GEN_TAC THEN CONV_TAC THEN PURE_ORCH_REWRITE_TAC[B3DIS_DEF]
    THEN [ S5 IS
      MATCH_MP_TAC GRAPHS_INSERT_EDGE THEN MATCH_MP_TAC GRAPHS_INSERT_EDGE
      THEN MATCH_MP_TAC GRAPHS_INSERT_VERTEX THEN IMP_RES_TAC CONNECTED_GRAPH;
      S6.2%

```

```

ASN_CAREN_TAC "p2 IS_VERTXES (H.Edge)" THENL[
% 5.2.1 p1 IS_VERTXES & /\ p2 IS_VERTXES & %
IMP_BES_TAC CONNECTED_GRAPH THEN IMP_BES_TAC V_INSERT_ANDOP
THEN ASN_REWRITE_TAC[VERTXES_INSERT_EDGE]
THEN REPEAT STRIP_TAC THEN IMP_BES_TAC CONNECTED_DEF
THEN EXISTS_TAC "1: (Part@Part@hbl)" THEN REPEAT CONJ_TAC
THEN ((REPEAT (MATCH_MP_TAC PATH_IS_EDGE) OR ELSE ALL_TAC)
THEN FIRST_ASSUME ACCEPT_TAC);
% 5.2.2 p1 IS_VERTXES & /\ "p2 IS_VERTXES & %
PONE_REWRITE_TAC[VERTXES_INSERT_EDGE; VERTXES_IS_IS_VERTXES]
THEN REPEAT STRIP_TAC THENL[
% 5.2.2.1 v1 = p2 /\ v2 = p2 %
POP_ASSUME MP_TAC THEN ASN_REWRITE_TAC[];
% 5.2.2.2 v1 = p2 /\ v2 IS_VERTXES & %
ASN_CAREN_TAC "v2 = (p1:Part)" THENL[
% 5.2.2.2.1 v2 = p1 %
EXISTS_TAC "[ (p1,p1,a2):Part@Part@hbl]" THEN REPEAT CONJ_TAC THENL[
% 5.2.2.2.1.1
MATCH_MP_TAC PATH_IS_EDGE THEN MATCH_MP_TAC PATH_IS_EDGE
THEN CONJ_TAC THENL[
MATCH_MP_TAC GRAPH_INSERT_VERTXES THEN IMP_BES_TAC CONNECTED_GRAPH;
ASN_REWRITE_TAC[VERTXES_IS_IS_VERTXES] THEN
CONV_TAC (ONCE_DEPTH_CONV STR_CONV) THEN FIRST_ASSUME ACCEPT_TAC];
% 5.2.2.2.1.2 %
ASN_REWRITE_TAC[PATH_ENTRY_SIMP];
% 5.2.2.2.1.3 %
ASN_REWRITE_TAC[PATH_EXIT_SIMP];
% 5.2.2.2.2 "v2 = p1 %
POP_ASSUME (\%. ASSUME_TAC (CONV_RULE(ONCE_DEPTH_CONV STR_CONV) e))
THEN IMP_BES_TAC CONNECT_LEMMA
THEN EXISTS_TAC "CONS (v1,p1,a2) (1: (Part@Part@hbl))" THEN
THEN REPEAT CONJ_TAC THENL[
% 5.2.2.2.2.1 %
SUBST1_TAC (ASSUME "v1 = (p2:Part)")
THEN MATCH_MP_TAC PATH_IS_EDGE THEN MATCH_MP_TAC PATH_IS_IS_CONS
THEN REPEAT CONJ_TAC THEN (FIRST_ASSUME ACCEPT_TAC OR ELSE ALL_TAC)
THEN CONV_TAC (ONCE_DEPTH_CONV STR_CONV) THEN FIRST_ASSUME ACCEPT_TAC;
% 5.2.2.2.2.2 %
PONE_REWRITE_TAC[PATH_ENTRY_CONS@e_arc] THEN REVL_TAC;
% 5.2.2.2.2.3 %
MAP_EVENTY IMP_BES_TAC [PATH_NOT_NULL; PATH_EXIT_CONS]
THEN SUBST1_TAC (IMPSC "(v1,p1,a2):Part@Part@hbl"
(ASSUME "1:(Part@Part@hbl). PATH_EXIT(CONS h 1) = PATH_EXIT 1"))
THEN FIRST_ASSUME ACCEPT_TAC];
% 5.2.2.3 v1 IS_VERTXES & /\ v2 = p2 %
ASN_CAREN_TAC "v1 = (p1:Part)" THENL[
% 5.2.2.3.1 v1 = p1 %
EXISTS_TAC "[ (p1,p2,a1):Part@Part@hbl]" THEN REPEAT CONJ_TAC THENL[
MATCH_MP_TAC PATH_IS_EDGE THEN CONJ_TAC THENL[
MAP_EVENTY MATCH_MP_TAC [GRAPH_INSERT_EDGE@GRAPH_INSERT_VERTXES]
THEN IMP_BES_TAC CONNECTED_GRAPH;
ASN_REWRITE_TAC[VERTXES_IS_IS_VERTXES; VERTXES_INSERT_EDGE];
ASN_REWRITE_TAC[PATH_ENTRY_SIMP];
ASN_REWRITE_TAC[PATH_EXIT_DEF; WALK_EXIT_DEF; e_doe=NULL];
% 5.2.2.3.2 "v1 = p1 %
POP_ASSUME (\%. ASSUME_TAC (CONV_RULE(ONCE_DEPTH_CONV STR_CONV) e))
THEN IMP_BES_TAC CONNECT_LEMMA

```

```

% TYPE EXISTS_TAC "APPEND 1 [(p1,p2,sl):Part*Part*Obj*sl]"
% THEN REPEAT OBJ_TAC THEN%L
% 6.2.2.3.2.1.1
% MATCH NP_TAC PATH APPEND THEN REPEAT OBJ_TAC THEN%L%L
% 6.2.2.3.2.1.2
% REPEAT (MATCH_NP_TAC GRAPHS_INSERT_EDGE)
% THEN MATCH_NP_TAC GRAPHS_INSERT_VERTEX
% THEN IMP_RES_TAC CONNECTED_GRAPH
% 6.2.2.3.2.1.3
% FORM_UBINVERT_TAC([HIS1,PATH_DEF]) THEN REPEAT OBJ_TAC THEN%L%L%L
% REPEAT (MATCH_NP_TAC PATH_INSERT_VERTEX THEN FIRST_ASSUM ACCEPT_TAC)
% MATCH_NP_TAC PATH_INSERT_EDGE THEN OBJ_TAC THEN%L
%   RAP_VERIFY MATCH_NP_TAC [GRAPHS_INSERT_EDGE,GRAPHS_INSERT_VERTEX]
% THEN IMP_RES_TAC CONNECTED_GRAPH
% ASS_UBINVERT_TAC[VERTEXES_INSERT_VERTEXES_VERTEXES_INSERT_EDGE]];
% FORM_OCR_UBINVERT_TAC([HIS1,LIST_CONV])
% THEN REINVERT_TAC([HIS1,LIST_CONV,OBS1,LIST_EMPTY])
% THEN IMP_RES_TAC PATH_REMOVE_INSERT_EDGE
% THEN POP_ASSUM (λ v. NOT_ASSUM)
% (CONTRAPRUS (ISPEC "(p1,p2,sl):Part*Part*Obj*sl" s))
% THEN POP_ASSUM MATCH_NP_TAC
% THEN RAP_VERIFY IMP_RES_TAC [CONNECTED_GRAPH,GRAPHS_INSERT_VERTEX_INSERT_EDGE]
% THEN POP_ASSUM MATCH_NP_TAC
% FORM_OCR_UBINVERT_TAC([HIS1,LIST_CONV])
% THEN REINVERT_TAC([DIS1,LIST_CONV;DIS1,LIST_EMPTY;V_1_DEF;v_2_DEF])
% THEN IMP_RES_TAC PATH_REMOVE_INSERT_VERTEX
% THEN POP_ASSUM (λ v. NOT_ASSUM) (CONTRAPRUS (ISPEC "p2:Part" s))
% THEN RAP_VERIFY IMP_RES_TAC [PATH_REMOVE_INSERT_VERTEX,UBINVERT,LIST]
% THEN SUBST1_TAC (ASSUME
%   "V1=LIST 1 (Part*Part*Obj*sl)list = CONJ(CONJ(sl))" (V_1 DEF L1))
% THEN POP_UBINVERT_TAC[REMOVE_DEF,DE_REMOVE_TAC]
% THEN DISCH THEN IMP_RES_TAC%L
% 6.2.2.3.2.1.3
% FORM_OCR_UBINVERT_TAC[PATH_INSERT,IMP]
% THEN OBJV_TAC (OCHR_DEPTH_CONV SYN_CONV)
% THEN FIRST_ASSUM ACCEPT_TAC%L
% 6.2.2.3.2.1.4
% FORM_UBINVERT_TAC[WH1,LIST_CONV;s,src;s_def,V_1_DEF]
% THEN SUBST5_TAC (CONV_BULK (OCHR_DEPTH_CONV SYN_CONV)
%   (ASSUME "v1 = PATH_VERIFY 1 (Part*Part*Obj*sl)list"))
% THEN REINVERT_TAC[REMOVE_DEF,DE_REMOVE_TAC]
% THEN OBJV_TAC THEN%L
% OBJV_TAC (OCHR_DEPTH_CONV SYN_CONV)
% SUBST4_TAC (CONV_BULK (OCHR_DEPTH_CONV SYN_CONV)
%   (ASSUME "v2 = (p2:Part*1)"))
% THEN FIRST_ASSUM ACCEPT_TAC%L
% 6.2.2.3.2.2
% IMP_RES_TAC PATH_INSERT_APPEND
% THEN POP_ASSUM (λ v. FORM_OCR_UBINVERT_TAC(v))
% THEN FIRST_ASSUM ACCEPT_TAC%L
% 6.2.2.3.2.3
% SUBST5_TAC (REINVERT_REMOVE[REMOVE]) (ISPEC "(p1,p2,sl):Part*Part*Obj*sl)"
%   (ISPEC "1:(Part*Part*Obj*sl)list" PATH_REMOVE_APPEND))
% THEN REINVERT_TAC[PATH_REMOVE_INSERT_REMOVE_INSERT_DEF;s_def,REMOVE]
% 6.2.2.4.1
% IF_VERIFY 1 (V_1 DEF L1) THEN%L N %

```

```
IMP_RHS_TAC CONNECTED_ONF THEN EXISTS_TAC "1:({Part@areSHib1})list"
THEN REPEAT CONJ_TAC THEN ((MAP_EVENT (\th REPEAT (MATCH_RP_TAC th))
[PATH_INS_EDGE,PATH_INS_VERTEX])) DEKLEK ALL_TAC)
THEN FIRST_ASSUM ACCEPT_TAC]]];];
```

```
class_theory();;
```

Appendix C

Listings of the verifier

This appendix lists the ML source files of the network verifier which consists of the following files:

rail.grm	the input grammar of the parser
rail.decls.ml	declarations of the parser generated by the parser generator
rail.ml	functions of the parser generated by the parser generator
rail.help.ml	functions used by the parser
rail.load.ml	loader of the verifier
ver.network.ml	verifier functions
mk.verifier.ml	source for creating the base theory in which the verifier works
Makefile	makefile for compiling the verifier

Since the files **rail.ml** and **rail.decls.ml** are generated automatically and very long, they are not listed.

C.1 The file rail.grm

```
FIRST_CHARS 'A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9'
CHARS 'A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 .'
RAIL_LOAD --> definition_part construction_part [END]
```

```

definition_part --> [DEFINITION] def_list.

def_list --> def def_list | [] .

def --> bpart | tpart | ppart | dpart
      | tcir | point | signal | edgejoin | edgeneg.

bpart --> [BPART] {def_bpart(TOKEN)}.

tpart --> [TPART] {def_tpart(TOKEN, TOKEN)}.

ppart --> [PPART] {def_ppart(TOKEN, TOKEN, TOKEN, point_conn)}.

dpart --> [DPART] {def_dpart(TOKEN, TOKEN, diam_conn, diam_conn)}.

point_conn --> [{} {get_point(TOKEN, TOKEN, TOKEN)} {}].

diam_conn --> [{} {get_diam(TOKEN, TOKEN)} {}].

tcir --> [TCIR] {def_tcir(TOKEN)}.

point --> [POINT] {def_point(TOKEN)}.

signal --> [SIGNAL] {def_signal(TOKEN, TOKEN)}.

edgejoin --> [EDGEJOIN] {def_ejoin(TOKEN, TOKEN)}.

edgeneg --> [EDGENEG] {def_eneg(TOKEN, TOKEN, TOKEN)}.

construction_part --> [CONSTRUCTION] simp_op net_op_list.

net_op_list --> net_op net_op_list | [] .

net_op --> njoin_op | edge_op.

simp_op --> [SIMP] {nb_simp(TOKEN)}.

njoin_op --> [NJOIN] {nb_njoin(part_unum, edge_unum)}.

edge_op --> [EDGE] {nb_edge(part_unum, edge_unum)}.

part_unum --> {get_parta(TOKEN, TOKEN)}.

edge_unum --> {get_edgen(TOKEN, TOKEN)}.

```

C.2 The file rail.help.ml

```

load_theory 'verifier';; %

let tcir_func = "tcir";;
let partop_func = "partop";;
let partec_func = "partec";;
let neg_func = "neg";;

```



```

let jaig_func = "J_sig";
let subsig_func = "Sub_sig";
let shsig_func = "Sh_sig";

let no_thm = TRUE;;

let is_upper s = X : string -> bool X
  let code = nascii_code s in
  let code_A = (nascii_code 'A') - 1 and code_I = (nascii_code 'I') + 1 in
  ((code > code_A) & (code < code_I));;

let is_lower s = X : string -> bool X
  let code = nascii_code s in
  let code_a = (nascii_code 'a') - 1 and code_n = (nascii_code 'n') + 1 in
  ((code > code_a) & (code < code_n));;

let telower s = X : string -> string X
  if (is_upper s)
  then
    let code = (nascii_code s) - (nascii_code 'A') in
    (nascii ((nascii_code 'a') + code))
  else s;;

let tempper s = X : string -> string X
  if (is_lower s)
  then
    let code = (nascii_code s) - (nascii_code 'a') in
    (nascii ((nascii_code 'A') + code))
  else s;;

let lower_string s = implode (map telower (explode s));;
let upper_string s = implode (map tempper (explode s));;

let is_digit s = X : string -> bool X
  let code = nascii_code s in
  let code_0 = (nascii_code '0') - 1 and code_9 = (nascii_code '9') + 1 in
  ((code > code_0) & (code < code_9));;

let is_num s = X : string -> bool X
  forall is_digit (explode s);;

let is_part s = X string -> bool X
  let al = (explode s) in
  (mem (hd al) ['B'; 'T'; 'D'; 'P']) & (forall is_digit (tl al));;

let is_ege s = X string -> bool X
  let al = (explode s) in
  (mem (hd al) ['j'; 'e']) & (forall is_digit (tl al));;

let mh_num s = X : string -> term X
  mh_const(s, "num");;

let def_point id = X string -> string list @ the X
  if (is_num id)
  then
    (let ptname = "P" ^ id in
     let t = mh_eq(mh_var(ptname, "Point"),

```

```

mh_comb( mh_comb( mh_comb("POINT", mh_const(id, ":"num")),
  getpos_func), getloc_func) in
  ([ptname], new_definition(ptname, t)))
else failwith 'expecting a number as point ID (def_point)';

let def_tcir id = % string -> string list @ thm %
if (is_num id)
then
  (let ptname = 'Q' ^ id in
   let s = mh_eq(mh_var(ptname, ":"tcir"),
    mh_comb(mh_comb("TCIR", mh_const(id, ":"num")), tcir_func) in
    ([ptname], new_definition(ptname, t)))
  else failwith 'expecting a number as circuit ID (def_tcir)';

let get_point (s1, s2, s3) = % (string @ string @ string) -> (string list @ thm) %
if (is_num s1) & (is_num s2) & (is_num s3)
then ([s1; s2; s3], no_thm)
else failwith 'expecting three numbers as adjacent part ID's (get_point)';

let get_dim (s1, s2) = % (string @ string) -> (string list @ thm) %
if (is_num s1) & (is_num s2)
then ([s1; s2], no_thm)
else failwith 'expecting two numbers as adjacent part ID's (get_dim)';

let def_hpart s = % string -> string list @ thm %
if (is_num s)
then
  (let ptname = 'H' ^ s in
   let s = mh_eq(mh_var(ptname, ":"Part"),
    mh_comb("HPART", (mh_num s))) in
    ([ptname], new_definition(ptname, s)))
  else failwith 'expecting a number as part ID (def_hpart)';

let def_tpart (id, tc) = % (string @ string) -> (string list @ thm) %
if ((is_num id) & (is_num tc))
then
  (let ptname = 'T' ^ id in
   let tcir = mh_const(("Q" ^ tc), ":"tcir") in
   let s = mh_eq(mh_var(ptname, ":"Part"),
    mh_comb( mh_comb("TPART", (mh_num id)), tcir)) in
    ([ptname], new_definition(ptname, s)))
  else failwith 'expecting two numbers as ID's (def_tpart)';

let def_ppart (id, cid, pid, ([trail, norm, rev], sb)) =
% (string @ string @ string @ (string list @ thm)) -> (string list @ thm) %
if (is_num id) & (is_num cid) & (is_num pid)
then
  (let ptname = 'P' ^ id in
   let tcir = mh_const(("Q" ^ cid), ":"tcir") in
   let pos = mh_const(("H" ^ pid), ":"Point") in
   let tri = mh_pair( (mh_num trail),
    mh_pair( (mh_num norm), (mh_num rev))) in
   let s = mh_eq(mh_var(ptname, ":"Part"),
    mh_comb( mh_comb(
      mh_comb("PPART", (mh_num id), tcir), pos), tri)) in
    ([ptname], new_definition(ptname, s)))
  else failwith 'expecting three numbers as ID's (def_ppart)';

```

```

let def_dpart (id, cid, ([pa1:pa2], th1), ([pb1:pb2], th2)) =
  λ : string & string & (string list & thm) & (string list & thm) ->
    (string list & thm) λ
  if (is_num id) & (is_num cid)
  then
    (let pname = 'D' ~ id in
     let scsr = mk_const (('D'~cid), "Tsr") in
     let l1 = mk_pair((mk_num pa1), (mk_num pa2)) in
     let l2 = mk_pair((mk_num pb1), (mk_num pb2)) in
     let t = mk_eq(mk_var(pname, "Pare"),
                  mk_comb(mk_comb(mk_comb("DP&T", (mk_num id)),
                                   scsr), l1), l2)) in
     ([pname], new_definition(pname, t)))
  else failwith "expecting two numbers as ID's (def_dpart)" ;;

let def_signal (id, stype) =
  λ : (string & string) -> string list & thm λ
  if (is_num id)
  then
    (let pname = 'S' ~ id in
     let rts = case stype of
       'RAIS'   => (mk_comb(mk_comb("SIGNAL", (mk_num id)), Raig_func)) |
       'RAIS_JOIN' => (mk_comb(mk_comb(
         mk_comb("SIGNAL", (mk_num id)), Raig_func), Jaig_func)) |
       'RAIS_OR'  => (mk_comb(mk_comb(
         mk_comb("SIGNAL", (mk_num id)), Raig_func), Skaig_func)) |
       'RAIS_OR_JOIN' => (mk_comb(mk_comb(mk_comb(
         mk_comb("SIGNAL", (mk_num id)), Raig_func), Skaig_func), Jaig_func)) |
       'SHUNT'   => (mk_comb(mk_comb("SIGNAL", (mk_num id)), Skaig_func))
     in
     let t = mk_eq(mk_var(pname, "Signal"), rts) in
     ([pname], new_definition(pname, t)))
  else failwith "expecting a number as ID (def_signal)" ;;

let def_ejoin (id, jtype) =
  λ : string & string -> string list & thm λ
  if (is_num id)
  then
    (let pname = 'J' ~ id in
     let jtp = mk_const (('J' ~ (lower_string jtype)), "Join") in
     let t = mk_eq(mk_var(pname, "Elbl"),
                  mk_comb("ELBL", jtp)) in
     ([pname], new_definition(pname, t)))
  else failwith "expecting a number as ID (def_ejoin)" ;;

let def_eig (id, jtype, sig) =
  λ : string & string & string -> string list & thm λ
  if (is_num id) & (is_num sig)
  then
    (let pname = 'e' ~ id in
     let jtp = mk_const (('J' ~ (lower_string jtype)), "Join") in
     let t = mk_eq(mk_var(pname, "Elbl"),
                  mk_comb(mk_comb("ELBLID", jtp), mk_const(('S'~sig), "Signal"))) in
     ([pname], new_definition(pname, t)))
  else failwith "expecting a number as ID (def_eig)" ;;

```

```

-----
% functions for construction part %
-----

letref rail,tmp,thm = TRUE;;

let mh_simp pt : (string list & thm) =
  let s = mh_const(pt, "Part") in
  let th = prove_simple_network s in
  ([pt], (rail,tmp,thm := save_thm((pt"YHM"), th))) ;;
% ([pt], save_thm((pt"YHM"), th)) ;; %

let mh_ujoin (([pt1; pt2], s1:thm), ([od1; od2], s2:thm))
% : (string list & thm) & (string list & thm) -> %
  (string list & thm) =
  let p1 = mh_const(pt1, "Part") in
  let p2 = mh_const(pt2, "Part") in
  let a1 = mh_const(od1, "Elbl") in
  let a2 = mh_const(od2, "Elbl") in
  let th = prove_network_ujoin rail,tmp,thm p1 p2 a1 a2 in
  ([pt1], (rail,tmp,thm := save_thm((pt1"YHM"), th))) ;;
% ([pt2], save_thm((pt2"YHM"), th)) ;; %

let mh_edge (([pt1; pt2], s1:thm), ([od1; od2], s2:thm))
% : (string list & thm) & (string list & thm) -> %
  (string list & thm) =
  let p1 = mh_const(pt1, "Part") in
  let p2 = mh_const(pt2, "Part") in
  let a1 = mh_const(od1, "Elbl") in
  let a2 = mh_const(od2, "Elbl") in
  let th = prove_network_edge rail,tmp,thm p1 p2 a1 a2 in
  ([pt1], (rail,tmp,thm := save_thm((pt1"YHM"), th))) ;;

let get_parts (s1, s2) = % : (string & string) -> (string list & thm) %
  if (is_part s1) & (is_part s2)
  then ([s1; s2], no_thm)
  else failwith "expecting two parts (get_parts)";;

let get_edges (s1, s2) = % : (string & string) -> (string list & thm) %
  if (is_edge s1) & (is_edge s2)
  then ([s1; s2], no_thm)
  else failwith "expecting two edges (get_edges)";;

```

C.3 The file rail.load.ml

% Generated parser load file

```

First load some basic definitions: %
loadf "/home/quail/hol/Library/parser/general";;

% Insert any other files you want loaded here: %
add_to_search_path "../signal/";;
add_to_search_path "../network/";;

```

```

load_library 'graph';

load_theory 'verifier';
unloaded_all 'NETWORK';

loadf 'var_network';
loadf 'rail_help';

% Now load the declarations: %
loadf 'rail_decls';

% Finally load in the function definitions: %
loadf 'rail';

let NPS = [ ('C', []); ('S', []) ];

let verify file_name =
  (new_theory file_name:
   let (name), th = PARSE_file((file_name ^ ".xle"), [], NPS) in
   (class_theory(); save_thm(name, th)) );

```

C.4 The file var_network.ml

```

%-----
% FILE: var_network.ml
% A railway network verifier
% AUTHOR: Nai Wang  DATE: 8 Jan 1992 %
%-----

%-----
% prove_simple_network = - : conv
% prove_simple_network p returns a theorem
% |- NETWORK (('P'), { })
% iff p is a part, s.e. w/ type :Part. %
%-----
let prove_simple_network p =
  SPEC p NETWORK_BIMP ? failwith 'prove_simple_network';

%-----
% prove_in_network = - : (term -> conv)
% prove_in_network p returns a theorem
% |- p IS_VERTEX =
% iff p is a part in the network =
%-----
let prove_in_network p = TAC.PROOF ([],
  "p IS_VERTEX "m",
  PURE_ORCH_MINIMITE_TAC[IS_VERTEX_DEF]
  THEN PURE_ORCH_MINIMITE_TAC[VERTICES]
  THEN CONV_TAC (IS_CONV ALL_CONV)) ? failwith ('not in network');

%-----
% pair_RQ_CONV = - : conv -> conv
% pair_RQ_CONV conv "(...x1,...)" = (...y1,...)" ==>

```

```

|- ((...xi,...) = (...yi,...)) = Y iff (xi = yi) for all i
|- ((...xi,...) = (...yi,...)) = F otherwise
case is used to prove (xi = yi), typically, it will be asm_eq_CONV
for fields of type num and bool_eq_CONV for type bool
-----
let pair_eq_CONV conv to =
  let ...[lhs:rhs] = strip_conv to in
  if (lhs = rhs) then (REWRITE_CONV REFL_CLAUSES sm)
  else
    (let lhs = TOP_DEPTH_CONV (REWRITE_CONV PAIR_EQ) to in
     (PUSH_REWRITE_RULE[ASM_CLAUSES;DEPT_CLAUSES]
      (RIGHT_CONV_RULE (DEPTH_CONV conv) lhs))) ?
  failwith 'pair_eq_CONV'
-----
1 Part_EQ_CONV = - : conv
Part_EQ_CONV "(xPART ...) = (x'PART ...)" ==>
  |- ((xPART ...) = (x'PART ...)) = Y
  iff the parts are syntactically identical
  |- ((xPART ...) = (x'PART ...)) = F otherwise 1
-----
let Part_EQ_CONV sm =
  let ...[lhs:rhs] = strip_conv to in
  let find_def s = if (is_const s) then
    definition (current_theory()) (for (dest_const s)) else TRUTH in
  if (lhs = rhs) then (REWRITE_CONV REFL_CLAUSES sm)
  else
    (let rhodef = find_def lhs in
     let rhodef = find_def rhs in
     let suhl = filter (\th.0. nat(th = TRUTH))
      [(rhodef.lhs):(rhodef.rhs)] in
     let exp = if not(suhl) suhl then
      SUBST_CONV suhl "lhs = 'rhs'" sm else (REFL sm) in
     let lhs',rhs' = dest_eq (and(dest_eq (concl exp))) in
     if (lhs' = rhs') then
       (PUSH_CONV_REWRITE_RULE [REFL_CLAUSES] exp)
     else
       (let suhl' = filter (\s. nat(s = TRUTH)) [(rhodef:rhodef) in
        let Part_distinct = theorem 'PART' 'Part_distinct' in
        let Part_all_distinct = append (CONJUNCTS Part_distinct)
          (exp (CONV_RULE (CONV_DEPTH_CONV SUBST_CONV)
            (CONJUNCTS Part_distinct))) in
        let Part_one_one = theorem 'PART' 'Part_one_one' in
        let thm = TAC_PROOF([()], "" 'sm'),
          (MAP_FIRST MATCH_ACCEPT_TAC Part_all_distinct)
          CONJLHS
          (PUSH_CONV_REWRITE_TAC[Part_one_one]
           THEN CONV_TAC (DEPTH_CONV asm_eq_CONV)
           THEN PUSH_REWRITE_TAC[ASM_CLAUSES;DEPT_CLAUSES])) in
        REFL_1STRNG thm)
      ? failwith 'Part_EQ_CONV'
-----
1 prove_not_in_network = - : (term -> conv)
prove_not_in_network p s returns a theorem
  |- p IS_WRONG s

```

```

iff p is not a part in the network n
-----
1
% prove_net_in_network p n = TAC.PROOF ([],
-- 'p IS_VERTEX "n"',
PROMO_CHECK_ADMINISTR_TAC[IS_VERTEX.CONV]
THEN PROMO_CHECK_ADMINISTR_TAC[VERIFY.CONV]
THEN CONV_TAC [EMPTY.CONV (IS_CONV PART_EQ.CONV))
THEN PROMO_CHECK_ADMINISTR_TAC[NOT_CLASHES]]
? failwith ('p is in network n');];

%-----
% prove_finite n = : conv
prove_finite "{ ... }" ==> [- FINITE { ... } %
%-----
let prove_finite tm =
let finite = mk_comb(mk_const('FINITE',
mk_type('fin'), [(type_of tm); "bool"]]), tm) in
SET_ELIM (FINITE.CONV finite) ?
failwith 'prove_finite: not a finite set';];

%-----
% SUC_CONV : conv
SUC_CONV "SUC n" ==> [- SUC n = n' where n' = n + 1 %
%-----
let SUC_CONV =
let check n = assert(!c. fun (dest, const c) => n) in
!n.
let .. atm = (check 'SUC' @ I) (dest, const tm) in
let nstr = string_of_int (int_of_string((fun c dest, const c) => 1) in
SVN (nm, CONV (mk_const(nstr, "nm"))) ?
failwith 'SUC_CONV';];

%-----
% CARD_CONV = : conv -> conv
CARD_CONV conv "CARD (n0,...,n(n-1))" ==> [- CARD (n0,...,n(n-1)) = n
false if the set cannot be proved to be finite.
conv is used by IS_CONV to check whether the new element is already
in the set. = g.
CARD_CONV nm, EQ_CONV "CARD (1,2,3)" ==> [- CARD(1,2,3) = 3
and
CARD_CONV nm, EQ_CONV "CARD (1,2,2)" ==> [- CARD(1,2,2) = 2 %
%-----
let CARD_CONV conv tm =
let comp = theorem 'nats' 'CARD_EMPTY' in
let cins = theorem 'nats' 'CARD_INSERT' in
let check n = assert(!c. fun (dest, const c) => n) in
letrec atrip_set in =
(let .. [a:] = (check 'INSERT' @ I)(atnip, const tm) in
h atrip_set a) ?
(fun (dest, const tm) => 'EMPTY' => [] | fail) in
let prove_finite tm =
let finite = mk_comb(mk_const('FINITE',
mk_type('fin'), [(type_of tm); "bool"]]), tm) in
SET_ELIM (FINITE.CONV finite) ?
failwith 'prove_finite: not a finite set' in
let .. cns = (check 'CARD' @ atrip_set) (dest, const tm) in
let .. [aty] = dest.type(type_of (rand tm)) in

```

```

let empty = mh_const('EMPTY', "(:empty)out") in
let ins = mh_const('INSERT', "(:ety->(:ety)out->(:ety)out") in
let ifn cith a (insc.ith) =
  (mh_comb (mh_comb(ins, a), insc),
   (let ifn = prove.Finite insc in
    let sh = CONJ_RULE ((INCR_SEPTS_CONV (IN_CONV (conv)))
    THENC (CONJ_SEPTS_CONV CONJ_CONV)) (SPEC a (MATCH_MP cith ifn)) in
    let sh' = PURE_CONJ_BNDRITE_RULE[ifn] sh in
    CONJ_RULE (CONJ_SEPTS_CONV SPEC_CONV sh') in
    and (failif (ifn cith) else (empty, conv)) ?
    failifith 'CARD_CONV'))

```

```

-----
% mh_out_list = - : ((term -> bool) -> term -> term list)
mh_out_list f as returns a list of elements which satisfies the
predicates f, i.e., [ x1 ] where x1 is as and f x1. %
-----

```

```

let mh_out_list f as v
let check at = assert((λc. fset(dest, count c) = at) in
letrec strip_out tm =
  (let ..[h;u] = (check 'INSERT' 0 1)(convr_comb tm) in
   h .. strip_out v) ?
  (fset(dest, count as) = 'EMPTY' -> [] | fail) in
let old = strip_out as in
let ..[sig] = deat_type(type, of as) in
let ifn P a = if (P a) then ( a .. a)
  else a in
letlist (ifn f) old [] ?
failifith 'mh_out_list';

```

```

-----
% mh_incident_out_list = - : (string -> term -> term list)
mh_incident_out_list str as v returns a list of edges which are in
the set as and are incident TO/FROM/WITH the given vertex v. %
-----

```

```

let mh_incident_out_list str as v =
  let Pfrom = (λv. fset(dest, pair v) = v) in
  let Pto = (λv. fset(dest, pair(snd (dest, pair v))) = v) in
  let Pwith = (λv. (fset(dest, pair v) = v) or
    (fset(dest, pair(snd (dest, pair v))) = v)) in
  let P = if (str = 'FROM') then Pfrom else
    if (str = 'TO') then Pto else
    if (str = 'WITH') then Pwith else
    failifith 'mh_incident_out_list: unknown type string' in
  (mh_out_list P as) ? failifith 'mh_incident_out_list';

```

```

-----
% prove_in_incident = - : term -> term -> conv
prove_in_incident a d v ==> [ - a IS (INCIDENT_TO d v) %
-----

```

```

let prove_in_incident a d = TAC_PROOF([(),
  "a IS INCIDENT_TO d v",
  PURE_CONJ_BNDRITE_TAC[INCIDENT_TO_DEF],
  THEN PURE_CONJ_BNDRITE_TAC[IS_EDGE_DEF],
  THEN PURE_CONJ_BNDRITE_TAC[INCID],
  THEN CONV_TAC INT_SPEC_CONV

```



```

THEN PURE_REWRITE_TAC[a,des,DEF;FST;END;REPL_CLAUSE;AND_CLAUSES]
THEN CONV_TAC (IS_CONV ALL_CONV);;

%-----%
% list_to_set = [] term list -> term
list_to_set ["x"] ==> "{ x }" %
%-----%
let list_to_set el ty =
  if null el then "(0:('ty))" else
  let sty = type_of(hd el) in
  let empty = mk_const('EMPTY', "":('sty)list) in
  let ins = mk_const('INSERT', "":('sty)->('sty)list->('sty)list) in
  let ifrs s = mk_comb(mk_comb(ins, s), el) in
  (iflist ifrs el empty) ? failwith 'list_to_set';;

%-----%
% prove_incident_subset = - : term -> term -> conv
prove_incident_subset d v ==> [- (INCIDENT_TO d v) SURSET "x"]
%-----%
let prove_incident_subset d v s =
  let imp3 = GRS.ALL (d) S (CONJUNCTS (SPEC.ALL IMP_CLAUSES)) in
  TAC.PACGP([[] , "(INCIDENT_TO d ~v) SURSET ~v"),
  PURE_REWRITE_TAC[INCIDENT_TO_DEF;IS_EDGE_DEF;EDGE;SURSET_DEF]
  THEN CONV_TAC (GRC.DEPTH_CONV SET_SPEC_CONV)
  THEN GCR_TAC
  THEN PURE_REWRITE_TAC[IS_INSERT;BIGNT_AND_OVER_OR]
  THEN STRIP_TAC
  THEN POP_ASSON RP_TAC THEN ASM_REWRITE_TAC[]
  THEN PURE_REWRITE_TAC[a,des,DEF;FST;END]
  THEN CONV_TAC (RATOS_CONV (GRC.DEPTH_CONV Part_EQ_CONV))
  THEN PURE_REWRITE_TAC[imp3] ?
  failwith 'prove_incident_subset';;

%-----%
% INCIDENT_TO_CONV = - : term -> conv
INCIDENT_TO_CONV d v ==> [- (INCIDENT_TO d v) = { ... }
  is works out the set of edges which are incident to v and returns
  a theorem asserting this fact. %
%-----%
let INCIDENT_TO_CONV d v =
  let E2G = and (dest, pair d) in
  let all = mk_incident_set_list 'TO' E2G v in
  let sty = type_of E2G in
  let els = list_to_set all sty in
  let cith = GRS.ALL (and (EQ_IMP_RULE (SPEC.ALL INSERT_SURSET))) in
  let eth = !SPEC "INCIDENT_TO d ~v" EMPTY_SURSET in
  let ifrs cith x eth =
    MATCH RP cith (CONJ) (prove_in_incident x d v) eth in
  let th1 = prove_incident_subset d v els in
  let th2 = (iflist (ifrs cith) all eth) in
  (MATCH RP SURSET_ANTISYM (CONJ) th1 th2) ?
  failwith 'INCIDENT_TO_CONV';;

%-----%
% prove_RFC = - : term -> conv
prove_RFC p B ==> [- RFC p B
  iff p is a node in B and it is not fully connected. %

```

```

%-----1
let prove_rpc p n =
  let inth' = INCIDENT_TO_CONV n p in
  let pdef = if (is_const p) then
    definition (current_theory()) (fst (destr_const p)) also TRUTH in
  let inth = if (is_const p) then
    HONS [pdef] inth' else inth' in
  TAC.PROOF([[],"RPC 'N 'p'"),
    (SUBST_TAC [pdef] ORKLES ALL_TAC) THEN PURE_ORCH_REWRITE_TAC[RFC_DEF]
  THEN PURE_ORCH_REWRITE_TAC[IS_UNSAFE_DEF]
  THEN PURE_ORCH_REWRITE_TAC[INTH]
  THEN ((PURE_ORCH_REWRITE_TAC[CLASH_EPRF])
    THEN CONV_TAC (ORCH.DEPTH_CONV o== CONV) THEN MATCH_ACCEPT_TAC LESS_0)
  ORKLES
  (CONV_TAC (ORCH.DEPTH_CONV (CARD_CONV (pair_EQ_CONV Part.EQ_CONV)))
    THEN CONV_TAC (REDEPTH_CONV o== CONV)
    THEN PURE_REWRITE_TAC[LESS_HOMO_EQ;LESS_TWR;REFL_CLAUSE;OR_CLAUSE]])
  ? failwith 'prove_rpc' ;;

%-----2
% prove_network_njoin = - : thm -> term -> term -> term -> thm %
% prove_network_njoin thm1 p1 p2 j1 j2 ----> %
% |- NETWORK (JOIN at p1 j1 ((p2).()) p2 j2) %
% |- NETWORK ((p2) INSERT (v8 at1)). %
% ((n1,n2,j1) INSERT (n2,n1,j2) INSERT (EN at1)) %
% thm1 == NETWORK at1 %
% p1 is a vertex in at1 %
% p2 is a vertex to be added to the network %
% j1 is the join of the edge (p1,p2) %
% j2 is the join of the edge (p2,p1) %
%-----2
let prove_network_njoin thm1 p1 p2 j1 j2 =
  let p,n1 = (destr_const (const1 thm1)) in
  if (fst('NETWORK' = (fst (destr_const p)))) then
    failwith 'not NETWORK theorem' else
  let lm = (SPEC "[n1] NETWORK JOIN) in
  let thm2 = prove_in_network p1 n1 in
  let thm3 = SUP_ELIM (Part.EQ_CONV "p1 = 'p2") in
  let thm4 = prove_rpc p1 n1 in
  let thm5 = prove_not_in_network p2 n1 in
  let thm6 = RP (RP (SPEC[lm,p2] NOT_VER_INP_RPC) thm1) thm5 in
  let ante = (CONJ thm2 (CONJ thm3 (CONJ thm4 thm6)) in
  let lm' = SPEC [p1,p2] (RP lm thm1) in
  let network_class thm =
    let njointhm = RP (SPEC [n1,p1;j1;p2;j2] JOIN_EPR)
      (CONJ thm2 thm6) in
    let th = PURE_ORCH_REWRITE_RULE[VERTICES_EDGE]
      (PURE_ORCH_REWRITE_RULE[njointhm] thm) in
    (CONV_RULE (DEPTH_CONV (UNION_CONV Part.EQ_CONV)) th) in
  let nth = (SPEC [j1;j2] (MATCH_RP lm' ante)) in
  network_class nth ?
  failwith 'prove_network_njoin';;

%-----3
% prove_network_edge = - : thm -> term -> term -> term -> thm
% prove_network_edge thm1 p1 p2 j1 j2 ---->
% |- NETWORK ((p2,p1,j2) INSERT_EDGE ((p1,p2,j1) INSERT_EDGE at1))

```

```

|- NETWORKE ((v8 n1), ((p2.p1.j2) INSERT (p1.p2.j1) INSERT (m n1)))
  th1 = NETWORKE n1
  p1 is a vertex in n1
  p2 is another vertex in n1
  j1 is the join of the edge (p1.p2)
  j2 is the join of the edge (p2.p1) &
-----1
let prove_network_edges th1 p1 p2 j1 j2 =
  let p.n1 = (dest_cann (concl th1)) in
  if (not ('NETWORKE' = (fst (dest_cann p)))) then
    failwith 'not NETWORKE theorem' else
  let ln = (SPEC "n1" NETWORKE_NJOIN) in
  let thn1 = prove_in_network p1 n1 in
  let thn2 = EQV_ULIN (Part_EQ_CONV "p1 = 'p2'" in
  let thn3 = prove_EPC p1 n1 in
  let thn4 = prove_in_network p2 n1 in
  let thn5 = prove_EPC p2 n1 in
  let nate = CHJ3 thn2 (CHJ3 thn3 (CHJ3 thn4 thn5)) in
  let ln' = SPEC1 [p1;p2] (MP ln thn1) in
  let network_cannos thn =
    let njointhn = MP (SPEC1 [n1;p1;j1;p2;j2] NJOIN_EPP1)
      (CHJ3 thn2 thn3) in
    let th = FOLDS_ONCE_NJOINTH_NNLS[VERTICES.NJOIN]
      (FOLDS_ONCE_NJOINTH_NNLS[njointhn] thn) in
    (CONV_NNLS (DEFTS_CONV (CHJ3_CONV Part_EQ_CONV)) th) in
  let ntb = (SPEC1 [j1;j2] (MATCH_MP ln' nate)) in
  network_cannos ntb &
  failwith 'prove_network_edges';;

```

C.5 The file mk_verifier.ml

```

new_theory 'verifier';

add_to_search_path './signal/';;
add_to_search_path './network/';;

load_library 'graph';;

new_parent 'NETWORK';;
new_parent 'SIGNAL';;
new_parent 'PART';;

let tcf = new_definition('tcf', 'tcf = (\n:mm. sleep)');;

let patpce = new_definition('patpce', 'patpce = (\n:mm. normal)');;
let patlce = new_definition('patlce', 'patlce = (\n:mm. free_move)');;
let Hsig = new_definition('Hsig', "H_sig = HSID toe_aspect (\n:mm. green)");;
let Jsig = new_definition('Jsig', "J_sig = JSID (\n:mm. Y)");;
let Subsig = new_definition('Subsig', "Sub_sig = SUBSID (\n:mm. sub_not_abov)");;
let Shsig = new_definition('Shsig', "Sh_sig = SUBTSID (\n:mm. sh.off)");;

close_theory();;

```

C.6 The file Makefile

```

# Generated parser Makefile

# Version of HCL to be used:
HCL=hcl

# General definitions for all generated parsers:
GENERAL=/home/qaill/hcl/library/parser/general

# Insert entries for user-defined stuff here:
# Remember to insert the appropriate dependencies and "load"'s below.
var_network.ml.o: var_network.ml
echo 'set_flag(''abort_when_fail'',true);;' \
'load_library 'graph';;' \
'add_to_search_path './network/';;' \
'add_to_search_path './signal/';;' \
'load_theory 'NETWORK';;' \
'anteload_all 'NETWORK';;' \
'compile 'var_network';;' \
'quit();;' | $(HCL)

rail_help.ml.o: rail_help.ml var_network.ml.o verifier.th
echo 'set_flag(''abort_when_fail'',true);;' \
'load ' $(GENERAL);;' \
'load_library 'graph';;' \
'add_to_search_path './network/';;' \
'add_to_search_path './signal/';;' \
'load_theory 'NETWORK';;' \
'anteload_all 'NETWORK';;' \
'load 'var_network';;' \
'load_theory 'verifier';;' \
'compile 'rail_help';;' \
'quit();;' | $(HCL)

verifier.th: mh.verifier.ml
rm -f verifier.th
echo 'set_flag(''abort_when_fail'',true);;' \
'load 'mh.verifier';;' \
'quit();;' | $(HCL)

# Now compile the declarations:
rail_decls.ml.o: rail_decls.ml rail_help.ml.o
echo 'set_flag(''abort_when_fail'',true);;' \
'load ' $(GENERAL);;' \
'load_library 'graph';;' \
'add_to_search_path './network/';;' \
'add_to_search_path './signal/';;' \
'load_theory 'NETWORK';;' \
'anteload_all 'NETWORK';;' \
'load 'var_network';;' \
'load_theory 'verifier';;' \
'load 'rail_help';;' \
'compile 'rail_decls';;' \
'quit();;' | $(HCL)

# Finally do the actual functions

```

```
rail.ml.s: rail.ml rail_decide.ml.s
echo 'set_flag('short_when_fail',true);;\'
'load 'S(CHUNKS);;\'
'load_library 'graph';;\'
'add_to_search_path './network';;\'
'add_to_search_path './signal';;\'
'load_theory 'NETWORK';;\'
'externalize_all 'NETWORK';;\'
'load 'net_network';;\'
'load_theory 'verifier';;\'
'load 'rail_help';;\'
'load 'rail_decide';;\'
'compile 'rail';;\'
'quit();;\' | $(RM)

all: rail.ml.s
echo '***> Parser "rail" built.'

clean:
rm *.o *.th
```

Appendix D

Routes and control tables

This appendix lists the ML source of the definitions described in Chapter 8.

```
add_to_search_path './graph/';;
add_to_search_path './signal/';;
load_library 'graph';;
load_theory 'NETWORK';;
autoload_all 'NETWORK';;
autoload_all 'graph';;

new_theory 'ROUTE';;
load_library 'misc_lists';;

%-----%
% Functions for finding routes %
%-----%
% T if e is a trailing edge, i.e., (e_dec e) is a facing point or E
% (e_arc e) is a trailing point %

let TRAILING_EDGE_DEF = new_definition('TRAILING_EDGE_DEF',
  "TRAILING_EDGE (e:PartPartSetlib) =
    ((IN_PPART (e_dec e)) /\
     (PART_ID (e_arc e) = PART_PPT_TRAILING (e_dec e))) /\
    ((IN_PPART (e_arc e)) /\
     (PART_ID (e_dec e) = PART_PPT_TRAILING (e_arc e))))";;

% T if e is a normal edge, i.e., (e_dec e) is a facing point or E
% (e_arc e) is a trailing point %
let NORMAL_EDGE_DEF = new_definition('NORMAL_EDGE_DEF',
  "NORMAL_EDGE (e:PartPartSetlib) =
    ((IN_PPART (e_dec e)) /\
     (PART_ID (e_arc e) = PART_PPT_NORMAL (e_dec e))) /\
    ((IN_PPART (e_arc e)) /\
     (PART_ID (e_dec e) = PART_PPT_NORMAL (e_arc e))))";;

% T if e is a reverse edge, i.e., (e_dec e) is a facing point or E
% (e_arc e) is a trailing point %
let REVERSE_EDGE_DEF = new_definition('REVERSE_EDGE_DEF',
  "REVERSE_EDGE (e:PartPartSetlib) =
```

```

((IS_PPART (a_dec a)) /\
 (PART_ID (a_src a) = PART_PRT_REVERSE (a_dec a))) /\
 ((IS_PPART (a_src a)) /\
 (PART_ID (a_dec a) = PART_PRT_REVERSE (a_src a)))";;;

% If a1 and a2 are the same leg of a diamond crossing %
let SAME_LEG_DEF = new_definition('SAME_LEG',
  "SAME_LEG (a1:Part@Part@Hbl) (a2:Part@Part@Hbl) =
    ((IS_DPART (a_dec a1)) /\ (IS_DPART (a_src a2))) /\
    ((a_dec a1) = (a_src a2)) /\
    (let dp = a_dec a1 and
      id1 = PART_ID (a_src a1) and id2 = PART_ID (a_dec a2) in
      (((id1,id2) = (PART_DIA1 dp)) /\ ((id2,id1) = (PART_DIA1 dp)) /\
       ((id1,id2) = (PART_DIA2 dp)) /\ ((id2,id1) = (PART_DIA2 dp)))))) /\
    ((IS_DPART (a_src a1)) /\ (IS_DPART (a_dec a2)) /\
     (a_src a1) = (a_dec a2)) /\
    (let dp = a_src a1 and
      id1 = PART_ID (a_dec a1) and id2 = PART_ID (a_dec a2) in
      (((id1,id2) = (PART_DIA1 dp)) /\ ((id2,id1) = (PART_DIA1 dp)) /\
       ((id1,id2) = (PART_DIA2 dp)) /\ ((id2,id1) = (PART_DIA2 dp))))))";;;

```

```

%-----%
% Definition of routes --- a route is a path and the successive %
% edge of a PPART or DPART must satisfy the following conditions %
%-----%

```

```

let ROUTE_TAIL_DEF = new_list_rec_definition('ROUTE_TAIL_DEF',
  "(ROUTE_TAIL [] = T) /\
  (ROUTE_TAIL (CONS (h:Part@Part@Hbl) s) =
    ((a = []) /\
     ((IS_PPART (a_dec h) =>
      ((TRAILING_EDGE h) =>
        ((NORMAL_EDGE (HD s)) /\ (REVERSE_EDGE (HD s))) |
        (TRAILING_EDGE (HD s))) |
      ((IS_DPART (a_dec h) => (SAME_LEG h (HD s)) | T)) /\
      (ROUTE_TAIL s))))";;;

```

```

let ROUTE_DEF = new_definition('ROUTE_DEF',
  "ROUTE (B:Setuch) (r:(Part@Part@Hbl)list) =
  (RETHROW B /\ (PATH B r) /\ (ROUTE_TAIL r) /\
   (IS_ELM_SIGNAL (a1b (HD r)) /\ (IS_ELM_SIGNAL (a1b (LAST r))))";;;

```

```

let ROUTE_EDGES_DEF = new_definition('ROUTE_EDGES_DEF',
  "ROUTE_EDGES (r:(Part@Part@Hbl)list) = (MUTLAST r)";;;

```

```

let ROUTE_PARTS_DEF = new_definition('ROUTE_PARTS_DEF',
  "ROUTE_PARTS (r:(Part@Part@Hbl)list) =
  VED_LIST (MUTLAST (VL r))";;;

```

```

%-----%
% Conflicting routes %
%-----%

```

```

let CONFLICTING_ROUTE_DEF = new_definition('CONFLICTING_ROUTE_DEF',
  "CONFLICTING_ROUTE (B:Setuch) r1 r2 =
  (ROUTE B r1) /\ (ROUTE B r2) /\
  "(DIR)_LIST (ROUTE_PARTS r1) (ROUTE_PARTS r2))";;;

```

```

%-----%
% Functions for proving routes %
%-----%
% TCIRCHITS returns a list of track circuits in the route %
let TCIRCHITS_DEF = new_definition('TCIRCHITS_DEF',
  "TCIRCHITS (x:(Part*Part*bl))list =
    MAP PART_CIRCUIT (ROUTE_PARTS x)";);

% BORN returns a list of points required NORMAL if a movement %
% from p1 to p2 is made. [] is returned if none is required %
let BORN_DEF = new_definition('BORN_DEF',
  "BORN (p1,p2:(bl*bl)) =
    ((IS_PPART p1) /\ (PART_PRT_NORMAL p1 = PART_ID p2)) =>
      [PART_POINT p1] | []";);

% BORN_POINTS returns a list of points required NORMAL in the route %
let BORN_POINTS_DEF = new_definition('BORN_POINTS_DEF',
  "BORN_POINTS x = FLAT (MAP BORN (ROUTE_EDGES x))";);

% REV returns a list of points required REVERSE if a movement %
% from p1 to p2 is made. [] is returned if none is required %
let REV_DEF = new_definition('REV_DEF',
  "REV (p1,p2:(bl*bl)) =
    ((IS_PPART p1) /\ (PART_PRT_REVERSE p1 = PART_ID p2)) =>
      [PART_POINT p1] | []";);

% REV_POINTS returns a list of points required REVERSE in the route %
let REV_POINTS_DEF = new_definition('REV_POINTS_DEF',
  "REV_POINTS x = FLAT (MAP REV (ROUTE_EDGES x))";);

% EXIT_SIGNAL returns the exit signal of a route %
let EXIT_SIGNAL_DEF = new_definition('EXIT_SIGNAL_DEF',
  "EXIT_SIGNAL (x:(Part*Part*bl))list =
    let a = alb (LAST x) in
    ((IS_BLK_SIGNAL a) => [BLK_SIGNAL a] | []););

% CONFLICT_ROUTE returns a list of routes which are in x and
% are conflicting routes with x %
let CONFLICT_ROUTE_DEF = new_list_rec_definition('CONFLICT_ROUTE_DEF',
  "CONFLICT_ROUTE [] x = [] /\
  "CONFLICT_ROUTE (CONS h x) x =
    "(DIRJ_LIST (ROUTE_PARTS x) (ROUTE_PARTS h)) /\
    "(h = x) =>
      (CONS h (CONFLICT_ROUTE x x)) | (CONFLICT_ROUTE x x)";);

% ENTRY_SIG returns the signal attached at the first edge of the route %
let ENTRY_SIG_DEF = new_definition('ENTRY_SIG_DEF',
  "ENTRY_SIG (x:(Part*Part*bl))list = let a = alb (IN x) in
  ((IS_BLK_SIGNAL a) => [BLK_SIGNAL a] | []););

let ENTRY_SIGNALS_DEF = new_definition('ENTRY_SIGNALS_DEF',
  "ENTRY_SIGNALS x = MAP ENTRY_SIG (CONFLICT_ROUTE x x)";);

let FILTER_DEF = new_list_rec_definition('FILTER_DEF',
  "FILTER [] (f=>bool) = [] /\
  "FILTER (CONS h x) f =
    (f h) => (CONS h (FILTER x f)) | (FILTER x f)";);

```



```

let CR_TARK_DEF = new_list_rec_definition('CR_TARK_DEF',
  "(CR_TARK [] (g:Part) = []) /\
  (CR_TARK (CONS h s) p =
    (h = p) => [] | (CONS h (CR_TARK s p)))");;

let CR_PRS_DEF = new_definition('CR_PRS_DEF',
  "CR_PRS (g:Part) p11 =
    let cr1st = FILTER p11 (\l. (ELLEN l p)) in
    MAP (\l. CR_TARK l p) cr1st");;

let CR_PTS_DEF = new_list_rec_definition('CR_PTS_DEF',
  "(CR_PTS [] (p11:((Part)list)list) = []) /\
  (CR_PTS (CONS (g:Part) s) p11 =
    ((IS_PPART p) /\ (IS_SPART p)) =>
      (APPEND (CR_PRS p p11) (CR_PTS s p11))
    | (CR_PTS s p11))");;

let CR_TCIRCUITS_DEF = new_definition('CR_TCIRCUITS_DEF',
  "CR_TCIRCUITS r v1st =
    let cr1st = CONFLICT_ROUTES v1st r in
    let p1st = PLAY (CR_PTS (ROUTE_PARTS r) (MAP ROUTE_PARTS v1st)) in
    (MAP PART_CIRCUIT p1st)");;

clear_theory();;

```

Appendix E

Level crossing—a case study

This appendix lists the ML source files of the level crossing case study. A level crossing is represented by an object of type :Cross consisting of five state functions: Highway whether highway traffic is occupying the crossing, Track whether rail traffic is occupying the crossing, Approaching whether there is a train approaching, HighwaySignal whether the highway stop signal is ON, and RailSignal whether the rail stop signal is ON. All of these functions return value of type :bool with the value T representing active state, e.g., in the case of Highway and Track state, T indicates the crossing is occupied by the traffic of the respective kind.

The state machine controlling the level crossing can be in one of the following internal states:

```
Road.Proc highway traffic can proceed;  
Rail.App a train is approaching;  
Road.Clr the crossing is clear of traffic;  
Rail.Set rail signal is set;  
Rail.Proc rail traffic can proceed;  
All.Lock both rail and highway traffic are stopped.
```

Initially, the machine is in the All.Lock state with both highway and rail signals proved ON. This is defined as the predicate INIT. The function NEXT specifies the possible state transitions. The crossing is safe if not both rail and highway traffics are allowed to proceed, i.e., not both the rail and highway signals are turned OFF. This is defined as the predicated SAFETY.

The theorem INIT.SAFE asserts that the initial state of the machine is safe. The theorem NEXT.SAFE asserts that, if the current state is safe, the next state of the machine is also safe. The theorem STATE.MACHINE.SAFE asserts that the state machine specified by INIT and NEXT possesses the safety property SAFETY.

```

new_theory 'CHRS';;
set_flag('stichy', true);;
new_params 'RS';;
outload_all 'LSA';;

let Cross_Axiom = define_type 'Cross_Axiom'
  'Cross = CHRS (sum->bool) (sum->bool) (sum->bool) (sum->bool)';;
% Highway Track Approaching HighwaySignal RailSignal %

let HIGHWAY_DEF = new_recursive_definition false Cross_Axiom 'HIGHWAY_DEF'
  "HIGHWAY (CHRS h t app sh st) = h";;

let TRACK_DEF = new_recursive_definition false Cross_Axiom 'TRACK_DEF'
  "TRACK (CHRS h t app sh st) = s";;

let APPROACH_DEF = new_recursive_definition false Cross_Axiom 'APPROACH_DEF'
  "APPROACH (CHRS h t app sh st) = app";;

let R_SIGNAL_DEF = new_recursive_definition false Cross_Axiom 'R_SIGNAL_DEF'
  "R_SIGNAL (CHRS h t app sh st) = sh";;

let B_SIGNAL_DEF = new_recursive_definition false Cross_Axiom 'B_SIGNAL_DEF'
  "B_SIGNAL (CHRS h t app sh st) = st";;

new_type_abbrev ('EState', "~:bool@bool@bool@bool@bool");;

let State_Axiom = define_type 'State_Axiom'
  'State = Head_Proc | Rail_App | Head_Clr | Rail_Set | Rail_Proc | All_Lock';;

let State_const_dist = save_thm('State_const_dist',
  prove_conststrs_distinct State_Axiom);;

let State_Induct = save_thm('State_Induct', prove_induction_thm State_Axiom);;

let State_cases = save_thm('State_cases', prove_cases_thm State_Induct);;

let CHRS_STATE = new_recursive_definition
  false Cross_Axiom 'CHRS_STATE'
  "CHRS_STATE (CHRS h tr a sh st) = (h v, tr t, a t, sh t, st t)";;

let SR_STATE = new_definition('SR_STATE',
  "SR_STATE ((ht:bool), (tr:bool), (st:bool), (sh:bool), (st:bool)) = sh");;

let ST_STATE = new_definition('ST_STATE', "ST_STATE (ht, trt, st, sh, st) = st");;

let R_STATE = new_definition('R_STATE', "R_STATE (ht, trt, st, sh, st) = sh");;

let TR_STATE = new_definition('TR_STATE', "TR_STATE (ht, trt, st, sh, st) = trt");;

let A_STATE = new_definition('A_STATE', "A_STATE (ht, trt, st, sh, st) = st");;

let INIT = new_definition('INIT',
  "INIT (a, s) = ((SR_STATE a) = T) /\ ((ST_STATE a) = T) /\ (a = All_Lock)");;

let SAFETY = new_definition('SAFETY',
  "SAFETY (a, s) = ~(SR_STATE a /\ ST_STATE a)");;

```

```

let NEXT = new_definition('NEXT',
  "NEXT (c, a) (c', a') =
    ((a = All_Lock) =>
      ((~(TS_STATE c /\ A_STATE c) => ((a' = Hand_Prac) /\ 'SE_STATE c' /\ ST_STATE c') |
        (A_STATE a => ((a' = Rail_App) /\ SE_STATE c' /\ ST_STATE a') |
          (a' = a) /\ (a' = c))) |
      ((a = Hand_Prac) =>
        (A_STATE c => ((a' = Rail_App) /\ SE_STATE c' /\ ST_STATE a') |
          (a' = a) /\ (a' = c)) |
        ((a = Rail_App) =>
          (~(SE_STATE c => ((a' = Hand_Clr) /\ SE_STATE c' /\ ST_STATE c') |
            (a' = a) /\ (a' = c)) |
            ((a = Hand_Clr) =>
              ((~(SE_STATE c /\ A_STATE c) =>
                ((a' = Rail_Sec) /\ SE_STATE c' /\ ST_STATE c') |
                  (a' = a) /\ (a' = c)) |
                ((a = Rail_Sec) =>
                  (TS_STATE c => ((a' = Rail_Prac) /\ SE_STATE a' /\ ST_STATE c') |
                    (a' = a) /\ (a' = c)) |
                    ((a = Rail_Prac) =>
                      (~(TS_STATE c => ((a' = All_Lock) /\ SE_STATE c' /\ ST_STATE c') |
                        (a' = a) /\ (a' = c)) |
                        (a' = a) /\ (a' = c)))))))))";);

let INIT_SAFE = prove_thm('INIT_SAFE',
  "(!c a. INIT(c, a) ==> SAFETY(c, a))",
  REPEAT_TAC[INIT.SAFETY] THEN REPEAT GEN_TAC THEN STRIP_TAC
  THEN ASM_REWRITE_TAC[]););

let NEXT_SAFE = prove_thm('NEXT_SAFE',
  "(!c a c'. NEXT (c, a) (c', a') /\ SAFETY (c, a) ==> SAFETY (c', a')",
  PUSH_ONCE REWRITE_TAC[INIT.SAFETY] THEN REPEAT GEN_TAC
  THEN REPEAT CHOOSE_TAC THEN STRIP_TAC THEN ASM_REWRITE_TAC[]););

let CROSS_MACHINE = prove_thm('CROSS_MACHINE',
  "(!c a. INIT(c, a) ==> SAFETY(c, a)) /\
  (!c a c'.
    NEXT (c, a) (c', a') /\ SAFETY (c, a) ==> SAFETY (c', a'))
  ==>
  ((c:num->State). LRA (INIT. NEXT) a = PLA$(INIT. SAFETY. NEXT) a )" ,
  STRIP_TAC THEN POP_ASSUM (MP_TAC <=> GEN_ALL a (SPEC1 ["(c:num->State) a";
    "(a:num->State) b"; "(c:num->State) (SUC b)"; "(a:num->State) (SUC b)"]])
  THEN POP_ASSUM (MP_TAC <=> GEN_ALL a (SPEC1 ["(c:num->State) a"; "(a:num->State) b"]])
  THEN REWRITE_TAC[LRA.PLA$] THEN REPEAT STRIP_TAC THEN RE_TAC
  THEN STRIP_TAC THEN REWRITE_TAC "a:num->State" THEN ASM_REWRITE_TAC[]
  THEN INDUCT_TAC THENL [ABS_TAC;
    ASSUM_LIKE (\abl. ASSUME_TAC (SPEC "a:num" (el 2 abl))) THEN ABS_TAC];);

% |- !c. LRA (INIT. NEXT) a = PLA$(INIT. SAFETY. NEXT) a
let CROSS_MACHINE_SAFE = save_thm('CROSS_MACHINE_SAFE',
  MP CROSS_MACHINE (CHOOSE INIT_SAFE NEXT_SAFE)););

close_theory(););

```

Appendix F

Dynamic networks and state machines

The following ML source listing contains the definitions described in Chapter 9.

F.1 The file dnetwork.ml

```
new_theory "DNET";;

new_type.abbrev ('SignalState', "(Aspect S bool S SubAspect) * SSubAspect");;
new_type.abbrev ('SignalStateFunc',
  "(num -> Aspect) S (num -> bool) S (num -> SubAspect) * (num -> SSubAspect)");;
new_type.abbrev ('PointState', "(Ppos S Ploc)");;
new_type.abbrev ('NetworkState',
  ":(Tstate)list S (PointState)list S (SignalState)list");;

let TC_STATE_FUNC_DEF = new_definition('TC_STATE_FUNC_DEF',
  "TC_STATE_FUNC (N:Network) =
  SET_LIST (IMAGE (TC_SFUNC o PART_CIRCUIT) (VS N))");;

let PST_STATE_FUNC_DEF = new_definition('PST_STATE_FUNC_DEF',
  "PST_STATE_FUNC (N:Network) =
  let plot = SET_LIST (IMAGE PART_POINT (VS N)) in
  (MAP (\p. (PST_POS p, PST_LOC p)) plot)");;

let SIG_STATE_FUNC_DEF = new_definition('SIG_STATE_FUNC_DEF',
  "SIG_STATE_FUNC (N:Network) =
  let signal = SET_LIST (IMAGE (ELINE_SIGNAL o elb) (EN N)) in
  (MAP SIG_SFUNC signal)");;

let APPLY_DEF = new_listing_definition('APPLY_DEF',
  "(APPLY s [] n = ([ ]:(ev)list)) /A
```

```

(APPLY (f succ) (s succ)) (COND (hd sl) (s s) =
  COND (f hd s) (APPLY f sl s))");;;

let APPLY_SIG_FUNC_DEF = new_definition('APPLY_SIG_FUNC_DEF',
  "APPLY_SIG_FUNC = (\u f s. ISL (sf:SignalStateFunc) =>
    ISL (((PST (OUTL sf)) s),
      ((PST (SND (OUTL sf))) s),
      ((SND (SND (OUTL sf))) s))
    | INR ((OUTL sf) s))");;;

let NETWORK_STATE_DEF = new_definition('NETWORK_STATE_DEF',
  "NETWORK_STATE (N:Network) s =
  let cstat = TC_STATE_FUNC N in
  let pstat = PST_STATE_FUNC N in
  let sstat = SIG_STATE_FUNC N in
  (APPLY (\u f s) cstat s),
  (APPLY (\u f s) pstat s),
  (APPLY APPLY_SIG_FUNC sstat s))");;;

let DNETWORK_DEF = new_definition('DNETWORK_DEF',
  "DNETWORK (N:Network) s = (VS N), ((s | (s IS_KNDR N) /\
  (IS_PPART (s_src s) =>
    ((PST_NORMAL (PART_POINT (s_src s)) s) /\
    (PART_ID (s_dst s) = PART_PST_NORMAL (s_src s)) /\
    (PST_REVERSE (PART_POINT (s_src s)) s) /\
    (PART_ID (s_dst s) = PART_PST_REVERSE (s_src s)) /\
    (PART_ID (s_dst s) = PART_PST_TRAILIN (s_src s)) |
  (IS_PPART (s_dst s) =>
    ((PST_NORMAL (PART_POINT (s_dst s)) s) /\
    (PART_ID (s_src s) = PART_PST_NORMAL (s_dst s)) /\
    (PST_REVERSE (PART_POINT (s_dst s)) s) /\
    (PART_ID (s_src s) = PART_PST_REVERSE (s_dst s)) /\
    (PART_ID (s_src s) = PART_PST_TRAILIN (s_dst s)) |
  T))"))");;;

% ROUTE_PROVED is T if there is a route from p1 to p2 in N and X
% X at time t, it can be proved
%

let ROUTE_PROVED_DEF = new_definition('ROUTE_PROVED_DEF',
  "ROUTE_PROVED r1 r s =
  let rstat = CONFLICT_ROUTE r1 r in
  (EVERY (\u. TC_CLEAR s s) (PCIRCUTS r)) /\
  (EVERY (\u. PST_NORMAL p s) (NORM_POINTS s)) /\
  (EVERY (\u. PST_REVERSE p s) (REV_POINTS r)) /\
  (EVERY (\u. (SIGNAL_FAIL s s) (REV_SIGNAL s)) /\
  (EVERY (\u. ON s s) (SETAT_SIGNAL s rstat)) /\
  (EVERY (\u. TC_CLEAR s s) (CONTCIRCUTS r rstat)))");;;

```

F.2 The file setlist.ml

```

new_theory 'setlist';

let ELN_EL_I1I = prove_thm('ELN_EL_I1I',

```

```

"if a. ELLEN 1 a ==> ?a. (a < LENGTH 1) /\ (a = EL a 1)".
LIST_INSERT_TAC THEN REMWRITE_TAC(ELLEN_DEF)
THEN REPEAT ONE_TAC THEN STRIP_TAC THENL[
  EXISTS_TAC "0" THEN POP_ASSUM SUBST1_TAC
  THEN REMWRITE_TAC(LENGTH_ELLEN_0_DEF);
  ONE_TAC THEN EXISTS_TAC "ONE a"
  THEN ASS. REMWRITE_TAC(LENGTH_EL.TL.LESS.ONE_DEF)];];

let LENLEN = TAC_PROOF([
  "in. (FINITE a) ==>
    (?a. (1a:0. (1a IN a = ELLEN (f a) a) /\ (CARD a = LENGTH (f a)))"),
  INT_INSERT_TAC THENL[
    EXISTS_TAC "\a:(0:1nat. []:(0:1)nat"
    THEN CONV_TAC (ONCE_DEPTH_CONV REW_CONV)
    THEN REMWRITE_TAC(OUT_IN_EMPTY.ELLEN_DEF.CARD_EMPTY.LENGTH);
    FIRST_ASSUM CHOICE_TAC THEN
    EXISTS_TAC "\a':(0:1)nat. (a' = a INSERT a) ==> CARD a (f a) | f a'"
    THEN CONV_TAC (ONCE_DEPTH_CONV REW_CONV) THEN CONJ_TAC THENL[
      REMWRITE_TAC(1a_INSERT.ELLEN_DEF) THEN ASS. REMWRITE_TAC();
      let is_set a = (fst(dst.cmb a) = "0") ? false in
      IMP_BES THEN (\a. FILTER_ASS. REMWRITE_TAC is_set (a) CARD_INSERT
        THEN FOLG_ONCE_ASS. REMWRITE_TAC(LENGTH) THEN REPT_TAC)];];

let TOLIST_DEF =
  % [- in. FINITE a ==>
    (1a. (1a IN a = ELLEN(TOLIST a a)a) /\ (CARD a = LENGTH(TOLIST a a))) %
    let lemma = CONV_RULE (ONCE_DEPTH_CONV RIGHT_IMP_EXISTS_CONV)
      THENC (REWRLEN_CONV) LENLEN;
    in
    new_specification 'TOLIST_DEF' ['constant','TOLIST'] lemma;];

let SET_LIST_DEF = new_definition('SET_LIST_DEF',
  "SET_LIST (a:(0:1)nat) = TOLIST a a");];

let SET_LIST_THM =
  % [- in. FINITE a ==>
    (1a. (1a IN a = ELLEN(SET_LIST a)a) /\ (CARD a = LENGTH(SET_LIST a))) %
    let l1 = CONV_RULE (ONCE_DEPTH_CONV STR_CONV) (SPEC_ALL SET_LIST_DEF)
    in
    prove_thm('SET_LIST_THM', FOLG_ONCE.REWRITE_RULE [l1] TOLIST_DEF);];

let ELLEN_SET_LIST = prove_thm('ELLEN_SET_LIST',
  "1a:(0:1)nat. FINITE a ==>
    (1a. ELLEN (SET_LIST a) a ==> ?a. (a < CARD a) /\ (a = (EL a (SET_LIST a))))".
  ONE_TAC THEN DISCH_TAC THEN IMP_ASS_TAC SET_LIST
  THEN POP_ASSUM SUBST1_TAC
  THEN ONE_TAC THEN DISCH_TAC THEN IMP_BES_TAC ELLEN_EL_1_DEF
  THEN EXISTS_TAC "a:0:1nat" THEN CONJ_TAC THEN FIRST_ASSUM ACCEPT_TAC);];

let ELLEN_SET_LIST_INSERT = prove_thm('ELLEN_SET_LIST_INSERT',
  "(a:(0:1)nat) a. FINITE a ==>
    (1a. 1a IN (a INSERT a) = ELLEN (SET_LIST (a INSERT a)) a",
  REPEAT ASS_TAC THEN STRIP_TAC THEN IMP_BES_TAC SET_LIST_INSERT
  THEN POP_ASSUM (\a. ASSUME_TAC (SPEC "a:a" a))
  THEN IMP_BES_TAC SET_LIST);];

let ELLEN_SET_LIST_EQ = prove_thm('ELLEN_SET_LIST_EQ',

```

```
Fig 6. (FINITE a /\ FINITE b) ==>
  ((a = s) = (b. KLEN (KEY_LIST a) a = KLEN (KEY_LIST b) b)),
  REPEAT STRIP_TAC THEN IMP_RES_TAC KEY_LIST
  THEN ASM_REWRITE_TAC[REVISION];;
```

Index

-->, 169, 170
-->>, 169, 170
<-->, 169, 170
>-->, 169, 170

Abbreviated types

Network, 81
ABS_Elbl, 200
ABS_Join, 195
ABS_Jsig, 186
ABS_MAspect, 186
ABS_Msig, 186
ABS_Mtype, 186
ABS_Part, 199
ABS_Ploc, 194
ABS_Point, 195
ABS_Ppos, 194
ABS_ShAspect, 185
ABS_Shsig, 185
ABS_Signal, 187
ABS_SubAspect, 185
ABS_Subsig, 185
ABS_Tcir, 195
ABS_Tatate, 195
APPLY, 141
APPLY_DEF, 141
BPART, 82, 199
BPART, 201
BPART_DEF, 201
clear, 75, 195, 197
clear_DEF, 187
CONFLICT_ROUTES, 133
CONFLICT_ROUTES_DEF, 133

CONFLICTING_ROUTES, 128
CONFLICTING_ROUTES_DEF, 128
CONNECTED, 72
CONNECTED_DEF, 72, 182
CONNECTED_GRAPH, 183
CONNECTED_IMS_EDGE, 184
CONNECTED_SING, 183
CR_PRS, 135
CR_PRS_DEF, 135
CR_PTS, 135
CR_PTS_DEF, 135
CR_TAKE, 134
CR_TAKE_DEF, 134
CR_TCIRCUITS, 136
CR_TCIRCUITS_DEF, 136

def_tpart, 118
DEGREE, 53, 172, 174
DEGREE_DEF, 53, 174
DELETE_EDGE, 55, 171, 173, 175
DELETE_EDGE_COMM, 57, 176
DELETE_EDGE_DEF, 55, 175
DELETE_INSERT_EDGE, 177
DELETE_VERTEX, 55, 171, 173, 175
DELETE_VERTEX_COMM, 58, 176
DELETE_VERTEX_DEF, 55, 175
DISJ_LIST, 67, 179, 180
DISJ_LIST_APPEND, 67, 180
DISJ_LIST_COMM, 67, 180
DISJ_LIST_CONS, 67, 180
DISJ_LIST_DEF, 67, 180
DISJ_LIST_EMPTY, 180
DISJ_PATH, 70, 182
DISJ_PATH_DEF, 70, 182
DNETWORK, 143

- DNETWORK.DEF, 143
double.yellow, 189
double.yellow, 78, 186
double.yellow.DEF, 189
double.yellow.flash, 78, 186, 189
double.yellow.flash.DEF, 189
DPART, 82, 200
DPART, 201
DPART.DEF, 201

E.ADJ.A, 54, 172, 174
E.ADJ.A.DEF, 54, 174
E.DELETE.ABSORP, 176
e.des, 172, 173
e.des, 175
e.des.DEF, 173
E.INSERT.ABSORP, 177
e.src, 172, 173
e.src, 175
e.src.DEF, 173
EDGE.EQ, 176
EDGE.IN.INSERT, 177
EDGE.IN.INSERT2, 177
EDGE.IN.INTER, 59, 177
EDGE.IN.UNION, 58, 178
EDGE.INSERT.EDGE, 177
EDGES, 51, 52, 175
EDGES.BETWEEN, 173, 174
EDGES.BETWEEN.DEF, 174
EDGES.INSERT.VERTEX, 177
EL.SET, 66, 179, 180
EL.SET.APPEND, 66, 180
EL.SET.DEF, 66, 180
elb, 172, 173
elb, 175
elb.DEF, 173
ELBL, 83, 200
ELBL, 202
Elbl.Axiom, 83, 202
Elbl.cases, 203
ELBL.DEF, 202
Elbl.distinct, 203
Elbl.Induct, 202
Elbl.ISO.DEF, 202

ELBL JOIN, 85, 200
ELBL JOIN, 202
ELBL JOIN.DEF, 85, 202
Elbl.one.one, 203
Elbl.SIGNAL, 85, 200
ELBL.SIGNAL, 202
ELBL.SIGNAL.DEF, 85, 202
Elbl.TV.DEF, 202
ELBLSIG, 83, 200
ELBLSIG, 202
ELBSIG.DEF, 202
ELEM, 65, 179, 180
ELEM.APPEND, 65, 180
ELEM.CONS, 65, 180
ELEM.DEF, 65, 180
ELEM.EL, 180
ELEM.IN.EL.SET, 66, 180
ELEM.NOT.UNIQUE.EL.CONS, 180
ENTRY.SIG, 134
ENTRY.SIG.DEF, 134
ENTRY.SIGNALS, 133
ENTRY.SIGNALS.DEF, 133
EQF.ELIM, 122
ES, 172, 174
ES.DEF, 174
EXIT.SIG, 133
EXIT.SIG.DEF, 133

faulty.aspect, 78, 186, 189
faulty.aspect.DEF, 189
FINITE.GRAPH, 52, 172, 174
FINITE.GRAPH.DEF, 52, 174
FINITE.GRAPH.INSERT.EDGE, 177
four.aspect, 78, 186, 190
four.aspect.DEF, 190
free.move, 76, 195, 196
free.move.DEF, 196
free.nor.rev, 196
free.nor.rev, 76, 195
free.nor.rev.DEF, 196
free.rev.nor, 196
free.rev.nor, 76, 195
free.rev.nor.DEF, 196
FUN.DEF, 170

- FUN.EMPTY.LEFT, 171
FUN.EMPTY.RIGHT, 171
FUN.I, 171
FUN.INV, 169, 170
FUN.INV.DEF, 170
FUN.INV.TY, 170
FUN.INVERSE, 170
FUN.INVERSE.DEF, 170
FUN.ISO.DEF, 170
FUN.ISO.o, 170
FUN.ONE.ONE.DEF, 170
FUN.ONE.ONE.o, 170
FUN.OWTO.DEF, 170
FUN.OWTO.o, 170
FUN.PINVERSE, 169, 170
FUN.PINVERSE.DEF, 170
FUN.TY, 170
- G.INS.INS.E, 178
G.INTER, 58, 171, 173, 175
G.INTER.ASSOC, 59, 177
G.INTER.DEF, 58, 175
G.INTER.IDENT, 59, 177
G.INTER.SYM, 59, 177
G.UNION, 57, 171, 173, 175
G.UNION.ASSOC, 58, 177
G.UNION.DEF, 57, 175
G.UNION.IDENT, 58, 177
G.UNION.INS.EDGES, 178
G.UNION.INSERT.EDGES, 178
G.UNION.SYM, 57, 177
GRAPH, 49, 172, 173
GRAPH.DECOMP, 175
GRAPH.DEF, 49, 173
GRAPH.DELETE.EDGE, 56, 176
GRAPH.DELETE.VERTEX, 56, 176
GRAPH.DIRECTED, 176
GRAPH.EDGE.VERTEX, 176
GRAPH.EQ, 175
GRAPH.EQUIV, 175
GRAPH.EXISTS, 49, 175
GRAPH.INSERT.EDGE, 56, 176
GRAPH.INSERT.EDGES, 178
GRAPH.INSERT.VERTEX, 56, 176
- GRAPH.INTER, 58, 177
GRAPH.ISO, 61, 173, 175
GRAPH.ISO.AUTO, 62, 179
GRAPH.ISO.DEF, 61, 175
GRAPH.ISO.SYM, 62, 179
GRAPH.ISO.SYM.INV, 62
GRAPH.ISO.SYM.INV, 179
GRAPH.ISO.TRANS, 62, 179
GRAPH.NOT.VERTEX.NOT.EDGE, 175
GRAPH.NOT.VERTEX.NOT.EDGE2, 176
GRAPH.PAIR, 175
GRAPH.UNION, 57, 177
green, 78, 186, 189
green.DEF, 189
green.flash, 189
green.flash, 78, 186
green.flash.DEF, 189
- HAS.LOOP, 51, 172, 174
HAS.LOOP.DEF, 51, 174
HAS.PATH, 182, 183
HAS.PATH.DEF, 183
- IGRAPH, 172, 174
IGRAPH.DEF, 174
IN.CONV, 122
IN.DEGREE, 53, 172, 174
IN.DEGREE.DEF, 53, 174
IN.ELEM, 65, 180
IN.INSERT.EDGE, 176
IN.INSERT.VERTEX, 176
INCIDENT.FROM, 52, 172, 174
INCIDENT.FROM.DEF, 52, 174
INCIDENT.TO, 172, 174
INCIDENT.TO.DEF, 174
INCIDENT.WITH, 52, 172, 174
INCIDENT.WITH.DEF, 52, 174
INCIDENT.WITH.INSERT.VERTEX, 176
INIT.DEF, 149
INSERT.DELETE.VERTEX, 177
INSERT.EDGE, 55, 89, 171, 173, 175
INSERT.EDGE.COMM, 56, 176
INSERT.EDGE.DEF, 55, 175
INSERT.VERTEX, 55, 89, 171, 173, 175
INSERT.VERTEX.COMM, 56, 176

- INSERT.VERTEX.DEF, 55, 175
 IS.BPART, 84, 200
 IS.BPART, 201
 IS.BPART.DEF, 201
 IS.DPART, 84, 200
 IS.DPART, 201
 IS.DPART.DEF, 201
 IS.EDGE, 171, 173, 174
 IS.EDGE.DEF, 174
 IS.ELBL.SIGNAL, 85, 200
 IS.ELBL.SIGNAL, 202
 IS.ELBL.SIGNAL.DEF, 85, 202
 IS.JCOND, 196, 198
 IS.JCOND.DEF, 198
 IS.JINSU, 196, 198
 IS.JINSU.DEF, 198
 IS.JOVER, 196, 198
 IS.JOVER.DEF, 198
 IS.JTERM, 196, 198
 IS.JTERM.DEF, 198
 IS.PPART, 84, 200
 IS.PPART, 201
 IS.PPART.DEF, 201
 IS.PRE.VER, 54, 172, 174
 IS.PRE.VER.DEF, 54, 174
 IS.SUC.VER, 54, 172, 174
 IS.SUC.VER.DEF, 54, 174
 IS.TPART, 84, 200
 IS.TPART, 201
 IS.TPART.DEF, 201
 IS.VERTEX, 122, 171, 173, 174
 IS.VERTEX.DEF, 174
 ISO.FINV, 171
 ISO.INVERSE, 171

 J.conduct, 74, 195, 198
 J.conduct.DEF, 198
 J.FUNC, 186, 188
 J.FUNC.DEF, 188
 J.insulate, 74, 195, 198
 J.insulate.DEF, 198
 J.overlap, 74, 195, 198
 J.overlap.DEF, 198
 J.terminate, 74, 196, 198

 J.terminate.DEF, 198
 Join.Axiom, 74, 199
 Join.cases, 199
 Join.const.dist, 198
 Join.INDUCT, 199
 Join.ISO.DEF, 197
 Join.TY.DEF, 197
 JSIG, 79, 186, 188
 Jsig.Axiom, 79, 192
 Jsig.cases, 192
 JSIG.DEF, 188
 Jsig.INDUCT, 192
 Jsig.ISO.DEF, 188
 Jsig.one.one, 192
 Jsig.TY.DEF, 188

 LEFT.FINV, 170
 LEFT.RIGHT.FINV, 171
 locked, 75, 195, 197
 locked.DEF, 197
 Logical types
 Elbl, 83, 87, 199
 Join, 74, 194
 Jsig, 79, 185
 Mispect, 78, 185
 Nsig, 78, 185
 Ntype, 78, 185
 Part, 82, 86, 199
 Ploc, 76, 194
 Point, 77, 194
 Ppos, 76, 194
 Shaspect, 80, 185
 Shsig, 80, 185
 Signal, 80, 185
 Subaspect, 79, 185
 Subsig, 79, 185
 Tcir, 75, 194
 Tetate, 75, 194
 LOOP, 51, 172, 174
 LOOP.DEF, 51, 174
 LSA, 146
 LSA, 146
 LSA.aq.PLSA, 146, 151
 LSA_imp.LSA, 147

- M.ASPECT, 186, 190
- M.ASPECT.DEF, 190
- M.FUNC, 186, 190
- M.FUNC.DEF, 190
- M.TYPE, 186, 190
- M.TYPE.DEF, 190
- MAIN_FAULTY, 79, 186, 190
- MAIN_FAULTY.DEF, 79, 190
- MAIN.OFF, 79, 186, 190
- MAIN.OFF.DEF, 79, 190
- MAIN.ON, 79, 186, 190
- MAIN.ON.DEF, 79, 190
- MAspect.Axiom, 78, 192
- MAspect.cases, 193
- MAspect.const.dist, 192
- MAspect.INDUCT, 193
- MAspect.ISO.DEF, 189
- MAspect.TY.DEF, 189
- mk_njoin, 119
- MK.SUBGRAPH, 61, 173, 175
- MK.SUBGRAPH.DEF, 61, 175
- MK.SUBGRAPH.GRAPH, 61, 179
- MK.SUBGRAPH.SUBGRAPH, 61, 179
- moving, 76, 194, 196
- moving.DEF, 196
- MSIG, 78, 186, 190
- Msig.Axiom, 78, 193
- Msig.cases, 193
- MSIG.DEF, 190
- Msig.INDUCT, 193
- Msig.ISO.DEF, 190
- Msig.one.one, 193
- Msig.TY.DEF, 190
- Mtype.Axiom, 78, 193
- Mtype.cases, 193
- Mtype.const.dist, 193
- Mtype.INDUCT, 193
- Mtype.ISO.DEF, 189
- Mtype.TY.DEF, 189
- MULTI.EDGE, 51, 172, 174
- MULTI.EDGE.DEF, 51, 174
- NETWORK, 88, 101, 116, 203
- NETWORK.BUFFER, 95, 204
- network.canon, 123
- NETWORK.CONNECTED, 100, 103, 204
- NETWORK.DEF, 88, 95, 115, 120, 203
- NETWORK.DIAM, 95, 204
- NETWORK.FINITE, 99, 204
- NETWORK.FINITE.GRAPH, 99, 204
- NETWORK.GRAPH, 98, 99, 100, 204
- NETWORK.INDUCT, 95, 204
- NETWORK.NJOIN, 96, 120, 204
- NETWORK.POINT, 95, 204
- NETWORK.SIMP, 120
- NETWORK.SIMP, 204
- NETWORK.STATE, 140
- NETWORK.STATE.DEF, 140, 141
- NETWORK.TRACK, 95, 204
- NFC, 89, 122, 203
- NFC.DEF, 89, 203
- NFC.SIMP, 204
- NJOIN, 90, 91, 93, 96, 100, 101, 203
- NJOIN.DEF, 90, 203
- NJOIN.EXP, 204
- NJOIN.EXP2, 204
- NORM, 132
- NORM.DEF, 132
- NORM.POINTS, 132
- NORM.POINTS.DEF, 132
- normal, 76, 194, 196
- normal.DEF, 196
- NOT.ELEM.UNIQUE_EL.CONJS, 180
- NOT.IN.SAME.GRAPH, 176
- NOT.IN.SAME.SET, 176
- NOT.NULL.VER.LIST, 180
- NOT.NULL.VER.LIST.CONJS, 181
- NOT.UNIQUE_EL.CONJS, 181
- NOT.VER.IMP.NFC, 204
- NOT.VER.INCIDENT.EMPTY, 176
- NOT.VERTEX.NOT.EDGE, 175
- NULL.GRAPH, 172
- NULL.GRAPH, 173
- NULL.MIL, 180
- NULL.NOT.ELEM, 65, 180
- occupied, 75, 195, 197
- occupied.DEF, 197

- OFF. 81, 187, 191
OFF_DEF. 81, 191
ON. 81, 187, 191
ON_DEF. 81, 191
OUT_DEGREE. 53, 172, 174
OUT_DEGREE_DEF. 53, 174
- PARSE.file, 117
Part.Axiom. 82, 202
Part.cases. 202
PART.CIRCUIT. 84, 200
PART.CIRCUIT. 201
PART.CIRCUIT_DEF. 201
PART.DIA1. 84, 200
PART.DIA1. 201
PART.DIA2. 84, 200
PART.DIA2_DEF. 201
PART.DIA_DEF. 201
Part.distinct. 202
Part.EQ_CONV. 122
PART.ID. 84, 200
PART.ID. 201
PART.ID_DEF. 201
Part.Induct. 202
Part.ISO_DEF. 200
Part.one.one. 202
PART.PNT.NORMAL. 84
PART.PNT.REVERSE. 84
PART.PNT.TRAILING. 84
PART.PNT.NORMAL. 200
PART.PNT.NORMAL. 201
PART.PNT.NORMAL_DEF. 201
PART.PNT.REVERSE. 200
PART.PNT.REVERSE. 201
PART.PNT.REVERSE_DEF. 201
PART.PNT.TRAILING. 200
PART.PNT.TRAILING. 201
PART.PNT.TRAILING_DEF. 201
PART.POINT. 84, 200
PART.POINT. 201
PART.POINT_DEF. 201
Part.TY_DEF. 200
PATH. 69, 182
PATH.APPEND. 71, 184
PATH.CAT. 184
PATH.CONNECTED. 183
PATH.COMS. 70, 184
PATH_DEF. 69, 182
PATH.EDGE.NO_LOOP. 184
PATH.ELEM.IS.EDGE. 184
PATH.ELEM.VERT.LIST.IS.VERTEX. 184
PATH.ENTRY. 69, 182
PATH.ENTRY.APPEND. 183
PATH.ENTRY.COMS. 183
PATH.ENTRY_DEF. 69, 182
PATH.ENTRY_SIMP. 183
PATH.EXIT. 69, 182
PATH.EXIT.APPEND. 183
PATH.EXIT.COMS. 183
PATH.EXIT_DEF. 69, 182
PATH.EXIT_SIMP. 183
PATH.G.UNION. 71, 184
PATH.GRAPH. 183
PATH.INS.EDGE. 72, 184
PATH.INS.EDGE2. 184
PATH.INS.INS.COMS. 184
PATH.INS.VERTEX. 72, 184
PATH.IS.EDGE. 184
PATH.IS.VERTEX. 184
PATH.NOT.NULL. 184
PATH.NOT.NULL. 183
PATH_SIMP. 184
PATH.TRAIL. 183
PATH.WALK. 183
PATH.WALK_ENTRY. 183
Ploc.Axiom. 78, 198
Ploc.cases. 198
Ploc.const.dist. 198
Ploc.INDUCT. 198
Ploc.ISO_DEF. 196
Ploc.TY_DEF. 196
PLSA. 146
PLSA. 146
PNT.ID. 195, 197
PNT.ID_DEF. 197
PNT.LOC. 195, 197
PNT.LOC_DEF. 197
PNT.NORMAL. 77, 195, 197

- PNT.NORMAL.DEF. 77, 197
PNT.POS, 195, 197
PNT.POS.DEF, 197
PNT.REVERSE, 77, 195, 197
PNT.REVERSE.DEF, 77, 197
PNT.RLOCKED, 195, 197
PNT.RLOCKED.DEF, 197
PNT.STATE.FUNCS, 139, 140
PNT.STATE.FUNCS.DEF, 139, 140
POINT, 77, 195, 196
Point.Axiom, 77, 198
Point.cases, 198
POINT.DEF, 196
Point.INDUCT, 198
Point.ISO.DEF, 196
Point.one.one, 198
Point.TY.DEF, 196
PPART, 82, 200
PPART, 201
PPART.DEF, 201
Ppos.Axiom, 76, 198
Ppos.cases, 198
Ppos.const.dist, 198
Ppos.INDUCT, 198
Ppos.ISO.DEF, 196
Ppos.TY.DEF, 196
PRE.VERS, 54, 172, 174
PRE.VERS.DEF, 54, 174
prove.in.network, 122
prove.network.edge, 120
prove.network.njoin, 119, 120
prove.simple.network, 120
prove.NFC, 122
prove.not.in.network, 122
PSUBGRAPH, 173, 175
PSUBGRAPH.DEF, 175
PSUBGRAPH.DELETE.EDGE, 178
PSUBGRAPH.DELETE.VERTEX, 179
PSUBGRAPH.IRREFL, 178
PSUBGRAPH.SUBGRAPH, 178
PSUBGRAPH.TRANS, 178
RED, 187, 190
red, 78, 186, 189
RED.DEF, 190
red.DEF, 189
remote.Locked, 196
remote.Locked, 195, 196
remote.Locked.DEF, 196
REP.Eibl, 200
REP.Join, 195
REP.Msig, 186
REP.MAspect, 186
REP.Msig, 186
REP.Mtype, 186
REP.Part, 199
REP.Ploc, 194
REP.Point, 195
REP.Ppos, 194
REP.ShAspect, 185
REP.Shaig, 185
REP.Signal, 187
REP.SubAspect, 185
REP.Subsig, 185
REP.Tcir, 195
REP.Tstate, 195
REV, 132
REV.DEF, 132
REV.POINTS, 132
REV.POINTS.DEF, 132
reverse, 78, 194, 196
reverse.DEF, 196
RIGHT.FINW, 170
ROUTE, 127, 130
ROUTE.DEF, 127, 130
ROUTE.EDGES, 128
ROUTE.EDGES.DEF, 128
ROUTE.PARTS, 128
ROUTE.PARTS.DEF, 128
ROUTE.PROVED, 145
ROUTE.PROVED.DEF, 145
ROUTE.TAIL, 127
ROUTE.TAIL.DEF, 127, 129, 130
SET.LIST.THM, 140
sh.faulty, 80, 185, 188
sh.faulty.DEF, 188
sh.off, 80, 185, 187

- sh.off.DEF, 187
- sh.on, 80, 185, 187
- sh.on.DEF, 187
- ShAspect.Axiom, 80, 192
- ShAspect.cases, 192
- ShAspect.const.dist, 192
- ShAspect.INDUCT, 192
- ShAspect.ISO.DEF, 187
- ShAspect.TY.DEF, 187
- ShSig.Axiom, 80, 192
- ShSig.cases, 192
- ShSig.INDUCT, 192
- ShSig.ISO.DEF, 188
- ShSig.one.one, 192
- ShSig.TY.DEF, 188
- SHUNT.FAULT, 185, 188
- SHUNT.FAULT.DEF, 188
- SHUNT.FUNC, 185, 188
- SHUNT.FUNC.DEF, 188
- SHUNT.NOT.ON.OFF, 194
- SHUNT.OFF, 185, 188
- SHUNT.OFF.DEF, 188
- SHUNT.ON, 185, 188
- SHUNT.ON.DEF, 188
- SHUNTSIG, 80, 185, 188
- SHUNTSIG.DEF, 188
- SIG.SFUNC, 187, 191
- SIG.SFUNC, 140
- SIG.SFUNC.DEF, 191
- SIG.STATE.FUNCS, 139
- SIG.STATE.FUNCS.DEF, 139, 140
- Signal.Axiom, 80, 193
- Signal.cases, 194
- SIGNAL.FAULT, 191
- SIGNAL.FAULT, 81, 187
- SIGNAL.FAULT.DEF, 81, 191
- SIGNAL.ID, 80, 187, 191
- SIGNAL.ID.DEF, 80, 191
- Signal.INDUCT, 194
- Signal.ISO.DEF, 190
- SIGNAL.JUNC, 187, 191
- SIGNAL.JUNC.DEF, 191
- SIGNAL.MAIN, 81, 187, 191
- SIGNAL.MAIN.DEF, 81, 191
- SIGNAL.NOT.ON.OFF, 82, 194
- Signal.one.one, 193
- SIGNAL.SHUNT, 187, 191
- SIGNAL.SHUNT.DEF, 191
- SIGNAL.STATE, 194
- SIGNAL.STATES, 82
- SIGNAL.SUB, 187, 191
- SIGNAL.SUB.DEF, 191
- Signal.TY.DEF, 190
- SIGNALM, 80, 187, 190
- SIGNALM.DEF, 190
- SIGNALMJ, 80, 187, 190
- SIGNALMJ.DEF, 190
- SIGNALMS, 80, 187, 191
- SIGNALMS.DEF, 191
- SIGNALMSJ, 80, 187, 191
- SIGNALMSJ.DEF, 191
- SIGNALS, 80, 187, 191
- SIGNALS.DEF, 191
- SIMPLE.GRAPH, 52, 172, 174
- SIMPLE.GRAPH.DEF, 52, 174
- SUB.FUNC, 185, 188
- SUB.FUNC.DEF, 188
- sub.not.show, 79, 185, 188
- sub.not.show.DEF, 188
- SUB.OFF, 185, 188
- sub.off, 79, 185, 188
- SUB.OFF.DEF, 188
- sub.off.DEF, 188
- SubAspect.Axiom, 79, 192
- SubAspect.cases, 192
- SubAspect.const.dist, 192
- SubAspect.INDUCT, 192
- SubAspect.ISO.DEF, 188
- SubAspect.TY.DEF, 188
- SUBGRAPH, 60, 173, 175
- SUBGRAPH.ANTISYM, 60, 178
- SUBGRAPH.DEF, 60, 175
- SUBGRAPH.DELETE.EDGE, 60, 178
- SUBGRAPH.DELETE.VERTEX, 61, 179
- SUBGRAPH.GRAPH, 60, 178
- SUBGRAPH.REFL, 60, 178
- SUBGRAPH.TRANS, 60, 178
- SUBSIG, 79, 185

Subsig.Axiom, 72, 192
Subsig.cases, 192
SUBSIG.DEF, 188
Subsig.INDUCT, 192
Subsig.ISO.DEF, 188
Subsig.one.one, 192
Subsig.TY.DEF, 188
SUC.VERS, 54, 172, 174
SUC.VERS.DEF, 54, 174

TC.CLEAR, 195, 197
TC.CLEAR.DEF, 197
TC.ID, 195, 197
TC.ID.DEF, 197
TC.LOCKED, 195, 197
TC.LOCKED.DEF, 197
TC.OCCUPIED, 195, 197
TC.OCCUPIED.DEF, 197
TC.SFUNC, 195, 197
TC.SFUNC.DEF, 197
TC.ST, 195, 197
TC.ST.DEF, 197
TC.STATE.FUNCS, 132
TC.STATE.FUNCS.DEF, 132
TCIR, 75, 195, 197
Tcir.Axiom, 75, 199
Tcir.cases, 199
TCIR.DEF, 197
Tcir.INDUCT, 199
Tcir.ISO.DEF, 197
Tcir.one.one, 199
Tcir.TY.DEF, 197
TCIRCUITS, 132
TCIRCUITS.DEF, 132
three.aspect, 78, 186, 190
three.aspect.DEF, 190
three.repeat, 78, 186, 190
three.repeat.DEF, 190
TL.VER.LIST, 181
TPART, 82, 89, 200
TPART, 201
TPART.DEF, 201
TRAIL, 68, 182
TRAIL.DEF, 68, 182

TRAIL.WALK, 183
Tstate.Axiom, 75, 198
Tstate.cases, 199
Tstate.const.dist, 198
Tstate.INDUCT, 199
Tstate.ISO.DEF, 197
Tstate.TY.DEF, 197
two.aspect, 78, 186, 190
two.aspect.DEF, 190
two.repeat, 78, 186, 190
two.repeat.DEF, 190

UNIQUE.EL, 68, 179, 180
UNIQUE.EL.APPEND, 181
UNIQUE.EL.CONS, 181
UNIQUE.EL.DEF, 68, 180
UNIQUE.EL.SIMP, 180
UNIQUE.EL.TL, 180
UNIQUE.EL.VER.LIST.TL, 181
UNIQUE.V.L.CONS, 181
UNIQUE.VER.LIST.APPEND, 181
UNIQUE.VER.LIST.CONS, 181

V.DELETE.ABSORP, 176
V.INSERT.ABSORP, 177
V.L, 68, 179, 180
V.L.APPEND, 68, 180
V.L.DEF, 68, 180
VER.ADJA, 53, 172, 174
VER.ADJA.DEF, 53, 174
VER.INCIDENT.NOT.EMPTY, 176
VER.LIST, 68, 179, 180
VER.LIST.APPEND, 68, 181
VER.LIST.CONS, 68, 181
VER.LIST.DEF, 68, 180
verify, 117
VERTEX.EDGE, 176
VERTEX.IN.INS.VERTEX, 177
VERTEX.IN.INTER, 52, 177
VERTEX.IN.UNION, 52, 178
VERTEX.INSERT.EDGE, 178
VERTEX.INSERT.VERTEX, 177
VERTICES, 51, 175
VERTICES.IN.UNION, 178
VERTICES.INSERT.EDGE, 177

THE BRITISH LIBRARY
BRITISH THESIS SERVICE

A Formal Theory of Railway Track Networks

TITLE In Higher-order Logic
and its Applications in Interlocking Design

AUTHOR Wai Wong

DEGREE

AWARDING BODY University of Warwick
DATE (1992)

THESIS
NUMBER

THIS THESIS HAS BEEN MICROFILMED EXACTLY AS RECEIVED

The quality of this reproduction is dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction.

Some pages may have indistinct print, especially if the original papers were poorly produced or if the awarding body sent an inferior copy.

If pages are missing, please contact the awarding body which granted the degree.

Previously copyrighted materials (journal articles, published texts, etc.) are not filmed.

This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no information derived from it may be published without the author's prior written consent.

Reproduction of this thesis, other than as permitted under the United Kingdom Copyright Designs and Patents Act 1988, or under specific agreement with the copyright holder, is prohibited.

1	2	3	4	5	6	REDUCTION X	20
cm						CAMERA	5
						No. of pages	

D175045