APPLICATIONS OF FORMAL METHODS IN ENGINEERING

Volume 1 of 2

Sang Cong Tran
B.Eng (Hons.), M.Phil (Eng.)

A thesis submitted for the degree of
Doctor of Philosophy in the Engineering Department
University of Warwick
December 1991

Abstract

The main idea presented in this thesis is to propose and justify a general framework for the development of safety-related systems based on a selection of criticality and the required level of integrity. We show that formal methods can be practically and consistently introduced into the system design lifecycle without incurring excessive development cost.

An insight into the process of generating and validating a formal specification from an engineering point of view is illustrated, in conjunction with formal definitions of specification models, safety criteria and risk assessments. Engineering specifications are classified into two main classes of systems, memoryless and memory bearing systems. Heuristic approaches for specification generation and validation of these systems are presented and discussed with a brief summary of currently available formal systems and their supporting tools.

It is further shown that to efficiently address different aspects of real-world problems, the concept of embedding one logic within another mechanised logic, in order to provide mechanical support for proofs and reasoning, is practical. A temporal logic framework, which is embedded in Higher Order Logic, is used to verify and validate the design of a real-time system. Formal definitions and properties of temporal operators are defined in HOL and real-time concepts such as timing marker, interrupt and timeout are presented. A second major case study is presented on the specification a solid model for mechanical parts. This work discusses the modelling theory with set theoretic topology and Boolean operations. The theory is used to specify the mechanical properties of large distribution transformers. Associated mechanical properties such as volumetric operations are also discussed.

Declaration

This dissertation is the result of my own work and, unless otherwise stated in the text, includes nothing which is the outcome of work done in collaboration. No part of this dissertation has already been, or is currently being, submitted for any degree, diploma or other qualification at any other university.

Acknowledgement

During the time of this project, it is pleasing to note the number of people who have given their help and support. Thanks are due to my supervisor Dr. Evor Hines. He diligently read through the various drafts of this thesis and made valuable written comments. His insistence on clear writing has set a standard which I will always try to aim for.

In addition, I would like to express my sincerest thanks to Professor John Cullyer. This work could never have taken shape without his continued guidance and support. His patience during my earlier days and his ever optimistic manner has brought me through some of the most difficult times. He made valuable suggestions on earlier drafts of this thesis, and was the source of many excellent discussions.

This work could not have started without the financial support of T&N Technology Limited, Goodyear Transformers Limited and the Science and Engineering Research Council. The technical advice and individual guidance received while working there greatly help in this project. Of the innumerable people who helped, I must explicitly acknowledge Kevin Marks at T&N Technology Ltd. and Ken Frewin at Goodyear Transformers Ltd., who were the source of many stimulating discussions.

Thanks are also due to colleagues who have been around at various times to ask questions or answer mine. I would particularly like to thank Wai Wong for help at various times with HOL theorems and tactics and in response to my persistent questions about LaTeX typesetting.

This work would not have been completed without the HOL system. Thanks are due to the hardware verification group at Cambridge University. I am especially grateful to Dr. Mike Gordon for leading the way in hardware verification and giving advice when it was most needed.

To my parents

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the cost associated with the development of computer-based systems continues to fall, industry worldwide is increasingly coming to depend upon new technology to maintain a competitive edge whilst being able to operate ever more complex control systems that require a high degree of reliability, integrity, dependability and safety. Such systems are generically referred to as being safety-critical.

Engineers and managers have begun to realise that with their new technology comes flexibility and additional safety considerations. These considerations include,

- The knowledge that the correct and safe operation of the business, capital equipment or control system is dependent upon such technology.
- The risk of causing a loss of human life, economic loss or environmental damage is outside their normal engineering judgement.
- That the conventional safety assessment are inefficient and fail to identify weaknesses in the operation of the system.

Till now the only assurance of correct operation of a safety-critical system has been by conventional testing techniques. However, such techniques are not without flaws and cannot locate all faults. Faults often only materialise many months or years after a system has been put into operation, possibly leading to very expensive recall procedure or retro-fits on site, if not dangerous incidents. Manufacturing and nuclear industry, as well as defence contractors, have an urgent requirement to gain confidence in the correct and safe operation of their computer based systems.

With the advent of computers, designers have been able to design more complex systems, since they can be simulated on computers and so debugged to some extent before manufacture. However, for large systems, exhaustive simulation is prohibitively expensive, and requires exponentially large simulation time even with today's fastest computers.

Due to these seemingly unsurmountable problems, designers have been forced to look in other directions to verify designs before building them. Even though the cost of manufacturing is coming down, it is still important to have confidence in the design before it is committed to implementation. This is essential since it is not always possible to test every aspect of the design in the time available. Whilst many applications can tolerate some errors, there are others where minor flaws in the system could mean dangerous or may be too expensive to tolerate. These include,

- **Safety critical applications :**
  Increasingly, integrated circuits of enormous complexity are being used in areas where errors in these circuits could lead to loss of life. Examples include flight control systems, railway signalling systems, electronic throttle controller, medical life-support systems such as pacemakers, military applications, nuclear plant controllers, and anti-lock braking systems in cars (see [35]).

- **Remotely sited applications :**
  Applications where access to the systems is difficult would benefit from having the design 100% correct. This is simply because the cost of repairing or replacing faulty components would be too high. Examples include systems installed in arctic regions, satellites, nuclear plants and systems installed on oil or gas pipelines.

- **Volume production applications :**

Many industries manufacture systems for the mass market. If the design of a device in such systems is faulty then the cost of recalling and repairing it would be very high. Examples include circuits in automobiles, in telecommunication systems, and so on.

With important applications such as these, it is necessary to ensure that a design is correct. To achieve this, attention has turned to formal methods for specifying and verifying the correctness of system design.

1.1 Background and Related Work

Techniques used for the complete specification and verification of systems are briefly summarised in this section. Summaries of only the most relevant work are given, together with pointers to others.

1.1.1 Design methodologies

To-date research reveals an increasing awareness of the process of design but is fundamentally naive in its treatment of methodologies and formal models. It is from work on programming methodology that the basic idea that we have used as the framework of this thesis, have been imported. There are a number of structured design methodologies available [105]. These include MASCOT, SSADM, and JSD. MASCOT is primarily a method for the design and implementation of real time systems. The method is widely used in the UK defence and avionics industries and originated from work done at RSRE. Whilst MASCOT is a development technique, which uses of a graphical data-flow network as the medium for expressing software structure, SSADM in contrast, is a well-defined step-by-step approach. Rather than concentrating on one approach such as functional analysis, SSADM regards functions and data with equal importance. Another important design methodology is the Jackson Development Method (JSD). Within JSD there is a distinction between specification and implementation. JSD approaches the problem via the idea of a model and its relation to function. First an abstract description is written and then one determines how that abstract description is created in an implementation. In JSD the abstract description is created in two steps, and its realisation in a third step which is the development procedure. It is not until the fourth step that system functions are considered.

1.1.2 Specification and verification for software

There are many formalism for the formal specification of software based upon axiomatic definition or model building. For example, the algebraic specification languages OBJ [40], and CLEAR [24] are axiomatic formalisms, and the notation Z [104] is model based. The semi-formal design methodology VDM (Vienna Development Method) is also model-based [67].

Research on specific specification techniques has, more recently, led to research into the general ideas about specifications and operations on specifications. These attempts at general theories aim at more complex notation than those given in Chapter 3 of this thesis, but they are relevant to the further development of the described framework and its application to system design. A particularly useful general study of refinement is described by Back [8].

Much work is concerned with the generation of logical approaches. For example, in Cohen [30], there are simple but general definitions concerning specifications modelled by sets of statements in formal systems. A simple study of operations on specifications is described by Sanella and Wirsing [100]. A substantial theory of such specifications is presented in Goguen and Burstall's theory of institutions [41]. Other approaches work on themes of specifications which are modelled by relations as in, for example [59], to analyse ideas about modularity such as in Back and Mannila [7], or algebraically, as in Bergstra [13].

1.1.3 Specification and verification for hardware

The formal verification of digital hardware is most simply founded upon clearly defined logics such as first order logic, higher order logic, temporal logic, modal logic and so on.

The early work of Milner and others on the LCF project [43] inspired much effort in the area of mechanised theorem proving. A specialised language was developed for specifying and verifying hardware [45] which led to the development of the interim LCF-LSM theorem proving system [46]. Many examples were done using this system including the verification of a simple computer [47]. Many improvements to the LCF system have been made over the years, including an improved rewriting package [91], and a new tactics package [92]. Hanna and Daeche then independently developed the VERITAS theorem prover based on higher order logic [53,54]. With the improved expressive power of higher order logic becoming increasingly attractive, and with the experience gained from the LCF family of theorem provers, Gordon then developed the HOL theorem proving system [49,50], which forms the basis of the work done in this thesis. A number of examples have been successfully completed which demonstrate the use of higher order logic as a vehicle for specification and verification. These include the verification at the detailed timing level of a D-flipflop implemented in logic gates (see [54] and [56] for proofs done in the VERITAS and the HOL systems respectively), the verification of a ring interface chip [48], re-proof of the computer in HOL [68], and the first level proof of correctness of the VIPER microprocessor [31].

The systems mentioned above are all based on general theorem provers, where the proof is done manually and the system merely does the housekeeping. In the direction of automated theorem provers is the work of Boyer and Moore [17]. This has been used by Hunt to verify the correctness of a 16-bit microprocessor [61]. The underlying logic of this theorem prover is first order predicate logic without quantifiers. In principle, proofs are done automatically by this system, but in practice the theorem prover needs to be guided considerably by carefully requesting simpler theorem to be proved first. Barrow's VERIFY system [12] is another example of an automated theorem prover. The underlying model used by this system is that of finite state machines, based on Gordon's LSM language [46]. In both of these systems, logic gates form the set of primitive devices on the basis of which hardware verification is done.

Another general formalism used for verification is temporal logic. An interesting variant is that developed by Moszkowski known as Interval Temporal Logic (ITL) [88]. Proofs using ITL were initially done by hand. Recent work by Hale [52] shows that the HOL system can be used to mechanise proofs in ITL. A subset of ITL has also been developed as an executable language known as Tempura [89]. More recently, Leeser has used a variant of this formalism together with Prolog to reason about circuits down to the detailed transistor level [72].

On the side of more specialised formalism are CIRCAL and μFP. The early work of Milne and Milner on Concurrent Processes [81] inspired a number of calculi. Milne went on to develop CIRCAL [82,83], while Milner went in a slightly different direction and developed the Calculus of Communicating Process (CCS) [85,86]. In the CIRCAL framework, the structure of hardware devices is represented hierarchically, communication between components being effected through commonly named ports. Behaviour in this framework is described as a sequence of events on the external ports of devices. Operators are provided for the composition and the hiding of ports. Several examples have been completed using this framework, including a simple CRT controller by Traub in [113]. Here Traub also presents the various temporal concepts needed to model different granularities of time and the means of moving between them. A Lisp based environment for doing proofs in this formalism has also been developed [112].

Another motivation for using formal methods is the ability to transform designs in a correctness preserving way and hence perform refinements. One common aim is to convert simple but inefficient algorithms into complex but efficient ones. A recent general treatment is presented by Sheeran [101]. Design transformation and derivations are emerging as a practical technique in research on special architectures such as systolic arrays as described in, for example [11].

Sheeran uses μFP [101], which is an extension of the programming language FP developed by Backus. Sheeran introduces the μ operator into the basic language FP to model memory in circuit. Both the behaviour (functional part) and the implementation (geometric part) of a circuit can be presented in this language. With the aid of the μ operator, the inputs and outputs are modelled as streams. Behavioural descriptions of circuits can be transformed into their geometric forms thus leading to correctness by construction. Only synchronous system can be modelled by this formalism. Examples done using this framework include the design of a systolic correlator [101].

The above work has concentrated on the gate level and higher. Winskel [116] described a compositional model for the more primitive components of a VLSI technology, namely transistors, capacitors, etc. This work is based on the simulation models developed by Bryant [21]. Though this work is at a more detailed level than what has been described so far, it is not clear how it can be related to the system level.

1.2 Overview of Thesis

Below is a brief description of the organisation of the material presented in this thesis.

- Chapter 2 provides a general framework for the development of safety-related systems based on the selection of criticality and the required level of integrity. Different development and validation techniques are proposed for different levels of integrity required. For very highly safety critical systems, the main emphasis is on the use of formal methods to verify the correctness and ensure that the required level of safety integrity is achieved. This chapter lays down a methodological approach based on which formal methods can be consistently introduced into, and incorporated with, the system design lifecycle.

- Chapter 3 gives an insight into the process of generating and validating a formal specification from an engineering point of view. We give formal definitions of safety and that of a specification. Engineering specifications are classified into two main classes of systems, memoryless and memory bearing systems. Heuristic approaches for specification generation and validation of these systems are presented and discussed. We also give a brief summary of currently available formal systems and their supporting tools and then the reasons for selecting higher order logic as a specification formalism are discussed.

- Chapter 4 provides a brief description of the HOL system. We describe the species of higher order logic used, the meta-language ML in which the logic is formulated, and the theorem proving strategies used to conduct proofs in HOL. The main emphasis in this chapter is on those features of the HOL system which are used in later chapters of this thesis.
- Chapter 5 illustrates the concept of embedding one logic within another mechanised logic in order to prove soundness and provide mechanical support for proofs and reasoning. This chapter provides a temporal framework based on which the design of a real-time system described in the following chapter can be rigorously specified and validated. Formal definitions and properties of temporal operators are defined in HOL and real-time concepts such as time marker, interrupt and timeout are also discussed.
- Chapter 6 first gives a brief overview of the Drive-by-Wire control system together with its interfaces. Then a novel design is presented which uses the temporal constructs of the previous chapter. A formal specification for this system is presented together with complete proof procedures of safety and correctness. Some of the difficulties involved in arriving at the correctness statement and constructing the proof are discussed.
- Chapter 7 provides a second case study in specifying a solid modelling theory for mechanical parts. The work illustrates one aspect of formal validation techniques, that is correctness by construction. This chapter presents the modelling theory with set theoretic topology and boolean operations. The theory is used in this case to specify the mechanical properties of large

distribution transformers. Associated mechanical properties of transformers such as volumetric operations are also discussed.

- Finally, Chapter 8 evaluates the research described in this thesis. Some ideas for further research, possible solutions to problems encountered, and improvements to current strategies are also proposed.

Chapter 2

Formal Methods in Safety-Related Systems

Despite the ever increasing importance of software systems in UK industry, there is considerable difficulty in providing systems that meet the user requirements, even when safety is not an issue. In response to increasing concern over the use of computer-based systems in safety-related applications, recent government reports and guidelines, such as the ACARD report [1] and the Draft Interim Defence Standard 00-55 [66], call for the use of formal methods, particularly for high integrity applications. Use of the best software development methods available, including formal methods, may also be advisable to prevent legal liability for damage or injury caused by malfunctioning software. However, despite these recommendation there is little evidence of these methods being adopted. Most industries using computer control seem reluctant to invest in staff and computer tools. This chapter explores the reasons for the special concerns for the safety of computer-based systems, the ways in which formal methods can reduce the risk of system failure, and how it can be effectively incorporated into the development cycle.

2.1 The Concept of Safety

It should be emphasised that the present research is concerned with systems which are used in *safety-related* applications. A safety-related system is one via which the safety of human beings and the environment is to be assured. Such safety-related systems can range from quite small items, such as traffic light controllers, to major applications such as nuclear power station and avionics systems. Therefore, safety-related systems should imply those which are critical in one or more of the following areas,

- **Safety critical:** a system in which failure or design fault could cause risk to human life.
- **Security critical:** a system in which failure or design fault could cause unauthorised disclosure, alteration, loss or destruction of material.
- **Performance critical:** a system in which failure or design fault could cause the system not to achieve an important objective.
- **Cost critical:** a system in which failure or design fault could cause a significant financial loss.
- **Environmental critical:** a system in which failure or design fault could cause significant damage to the environment or property. This could have associated with it, disruption to normal life and possibly significant political factors.

2.1.1 Safety in computer-based systems

There are many factors contributing to concern over the safety of computer-based systems. Firstly being digital, they are unable to respond in a continuous and interpolative manner that is characteristic of analogue solutions. As a result, computer-based system are unable to average out the effects of inherent design errors, or of erroneous data input. Consequently, digital design principles often gives way to rule of thumb and engineering judgement to achieve the desired solution.

A second feature of computer-based systems is that where software is used to carry out system functions there is a tendency to build more and more complex systems since constraints such as volume, weight, and hardware rework costs have lesser impact. As systems become more complex the ability to demonstrate adequately the correct performance of the system under all operating conditions is greatly reduced and there is a significant risk that design errors will not be detected. Testing systems and their components does not guarantee that they will operate safely with values between input limits, and furthermore an exhaustive testing of all possible paths through a complex system is practically impossible.

Additionally, concern for the safety of computer-based systems has been compounded by the rapid increase in the demand for computer skills in recent years. As a result many system designers have little training in the pure mathematics disciplines of set theory and logic. These factors have produced a generation of computer professionals who are almost unaware of the relevant mathematics and modelling techniques.

2.1.2 Applying formal methods

The design of a computer-based system, as for all sequential systems is based theoretically on propositional logic and on the assumption that system outputs and internal stages are a function of all previous inputs and the most recent internal state. For relay logic, this type of system has been traditionally represented through the use of state transition tables and diagrams. Unlike relay logic, however, a computer-based system does not behave in a concurrent fashion. For this reason, the order in which the logic is executed is identified as a logical relationship between input condition, system state and output condition for all values of input data. This is achieved through the use of predicate calculus, which allows properties of objects manipulated by the system to be expressed.

Formal methods are mathematical languages which possess rigorously defined syntax and semantics, based on the mathematics of set theory and logic. Using these methods a functional specification of a system can be regarded as a set of mathematical equations and the rules for logical equivalence can be used to prove that two expressions are equivalent. This process has the result that abstract specifications can be defined in the first instance to model the behaviour of the system as required by the user, and can be subsequently refined to more detailed expressions, and eventually into a suitable implementation. At each step logical equivalence can be proven between the expressions, thus proving that the final implementation complies with the abstract specification in every feature. The process offers a potential for eliminating the risk of failing to meet the user requirement, since specification can take place at an abstract level, and could potentially improve design efficiency, as the process is amenable to automation.

However, the use of formal mathematical techniques are not without its drawbacks. Therefore, in order to make an effective approach towards system development for a wide range of safety-related applications, a framework is needed to select and apply appropriate techniques during the development process. The following sections described a methodological approach in the development cycle of safety-related systems.

2.2 Design Methodology

The design approach taken in this thesis consists of a number of consecutive conceptual stages as follows,

- **Safety criteria selection:** The safety criteria are established in discussion with the user. These are the criteria by which the system is expected to conform for it to be judged operable. The safety criteria differ for each system, industry sector and application because of the way in which the system is to be used, given legal, economic and environmental considerations.

- **Determination of assessment levels:** For each hazard criteria a specific assessment level is established after discussion with the user. Since there are no universal techniques for making this type of assessment the criticality of the system with respect to the particular application, is determined and from this the appropriate technique is chosen. By this consultation the user is assured that the correct emphasis is placed on the application.

- **Selection of development techniques:** Each hazard criteria is evaluated using the selected technique, in terms of the hardware, software and as a complete system.
- **Validation and verification:** For each hazard criteria, the system will be assessed to conform to some recognisable standard of safe/reliable operation for the agreed assessment level or will be found to be deficient in some way.

2.2.1 Development lifecycle

The principles just described can be embodied in four main steps, as shown in Fig.2.1. The first two steps do not deal directly with the design process. Rather they are part of the preparation needed before one can design safety features into the target systems. They usually require a broad range of inputs, including some specialised plant knowledge. It should be noted that the final specification of the target system must itself be safe.

- **Step 1:** Safety relevant information on the plant and the environment is compiled in this activity. Sources of information can be plant and environment descriptions, risk analyses etc. Auxiliary information such as regulations and general safety criteria should also be gathered in this step. The result of this step is a set of safety goals which should be fulfilled in the design of the target system. These safety goals are often expressed as a set of unsafe states the plant should not be brought into by the target system. A set of specific design constraints is produced during this step, along with a partial validation plan.

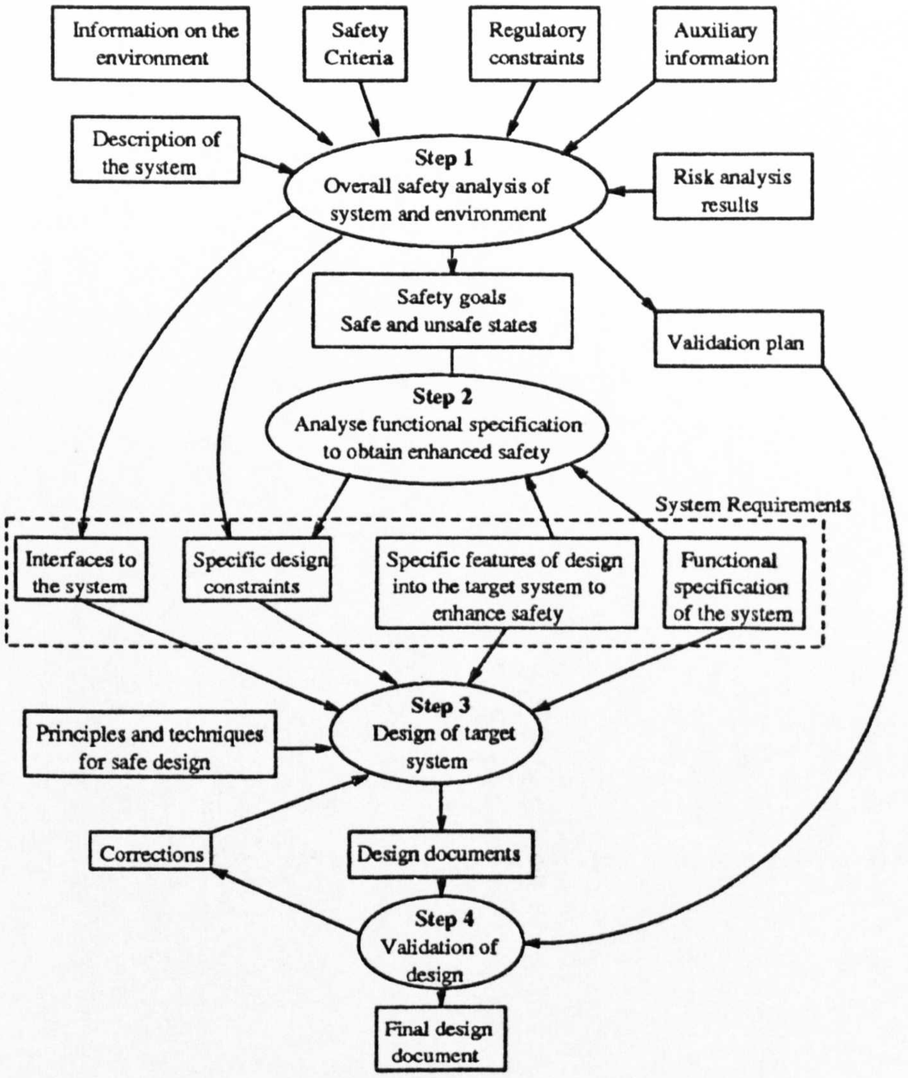

Information on the environment
Safety Criteria
Regulatory constraints
Auxiliary information
Description of the system
Step 1
Overall safety analysis of system and environment
Risk analysis results
Safety goals
Safe and unsafe states
Validation plan
Step 2
Analyse functional specification to obtain enhanced safety
System Requirements
Interfaces to the system
Specific design constraints
Specific features of design into the target system to enhance safety
Functional specification of the system
Step 3
Design of target system
Principles and techniques for safe design
Corrections
Design documents
Step 4
Validation of design
Final design document

Figure 2.1: Design procedures

- **Step 2:** The initial functional specification should be examined closely with respect to the safety goals established in step 1. The potentially safety critical parts of the target system should be highlighted in order to identify where specific safety measures should be designed into the target system. The result of this step is a specification of the special features which should be designed into the target system in order to enhance safety. These should be added to complete the functional specification.

The third and fourth steps are concerned with the design of the target system and its verification.

- **Step 3:** The target system is designed on the basis of the functional specifications and the description of the interfaces to the plant, as well as the specification of safety enhancing features made in step 2 and the specific design constraints determined in step 1. During this activity one should also utilise general principles and techniques for safe design, which should have been identified before the design started. The result of this step will be the target system design document.

- **Step 4:** The design documents are verified with respect to the functional specification, and it is also confirmed that the specific safety relevant requirements and recommendations developed in the first three steps have been followed. Information on any required corrections is returned to step 3. After this step, the final design document is complete.

Needless to say, a high quality of project management and quality control is needed in carrying out these four steps if the potential safety gain is to be realised. The verification procedure in the final step can only proceed properly if full records exists of the progress through the first three steps. The choice of suitable techniques for these steps and the application of the chosen ones are not trivial and the criteria for selecting appropriate techniques are discussed in section 2.5. The following sections discuss the principles and procedures involved in these steps in more details.

2.3 Safety Criteria Selection

The safety criteria for a system must satisfy any relevant legal requirements. Many regulatory authorities have issued guidelines establishing criteria and there are a number of national and international standards. In addition, many companies have developed their own internal standards. Table 2.1 shows some national

	Defence	Aerospace	Nuclear	Transport	Medical	Chemical	Automation	Oil/Gas	Energy	Finance
BS5750 [22]		√	√	√	√	√	√	√	√	√
BS5724					√					
BS5882 [23]	√		√							
IEC 880 [62]			√							
IEC SC65A WG9/10 [63]				√		√	√	√	√	
AQAP 1/13 [4,5]	√		√							
Def Stan 00-16 [36]	√		√							
Def Stan 00-55 [66]	√	√	√							
Def Stan 00-31 [65]	√	√								
CAA 690		√								
NES 620 [90]	√		√							
RTCA DO-178A [98]		√								
HSE PES1/2 [60]			√	√	√	√	√	√	√	

Table 2.1: Available standards (√ = applicable).

and international standards and their suitable application areas. It should be noted that the table is by no means complete.

From an engineering viewpoint, safety criteria for a system can be either qualitative, for example build to a design standard, or quantitative, such as to build a system so that it does not fail in a dangerous mode more than 10^{-6} times per year. In both cases, it is important to carry out hazard analysis either as part of a process to demonstrate compliance with a numerical target or to alert the designer as to whether or not the applied standard is appropriate.

Quantitative safety criteria are most often used for high risk situations but it should be noted that it is not possible, given the current state of knowledge, to quantify system safety integrity. However, this does not mean that quantified safety criteria for the system as a whole cannot be used. The numerical figure can be used to aid in the decision on the degree of rigour required in the system development process.

2.3.1 Measures of risk

Any approach to safety-related systems has to include a wide range of different types of application presenting a wide variety of different risks. The hazards associated with such diverse applications vary greatly both in character and consequence. Risk is defined by IEC in [63] as follows,

> *The combination of the frequency, or probability, and the consequence of a specified hazardous event.*

However, the means of combination of the frequency, or probability, and the consequence is undefined. The following means of combination could be considered given suitable values for the consequences of hazards,

$$H_n = \text{frequency of occurrence} \times \text{consequence of hazard}$$

where H_n represents some subjective assessed measure of risk associated with the hazard. As a result the degree of risk associated with different applications also varies. There is no clear cut point at which we can say this application is safety related and we shall apply full rigour to its development. Safety is essentially a concept which depends on current social values and in fact, it concerns with the level of risk at which the user is prepared to tolerate. Considerations which affect the acceptable level of risk H_n include

- the number of people affected when a hazard occurs, and
- the degree to which individuals may choose to subject themselves to the risk. For instance, it can be chosen subjectively and quantitatively that the affected radius of a railway accident is 50 metres, whilst it is about 20 metres in a car accident [107,108].

Moreover, since other economic and social factors also play an important part in the consideration, a faithful assessment of the level of risk involved should not only consider the negative hazardous effects but also the positive contributions

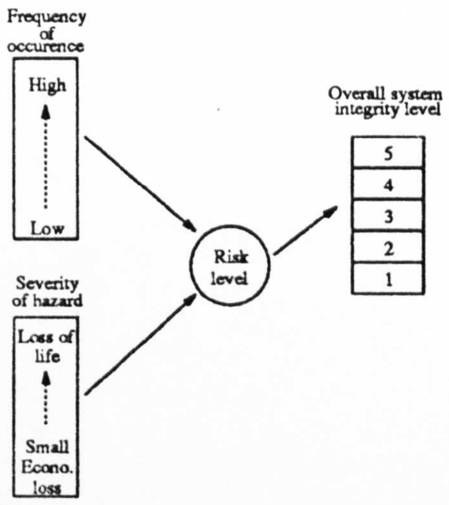


Figure 2.2: IEC mapping for software integrity level

of the system. That is the benefits to be derived from the application should also be taken into account.

The benefits derived from the application and harmful consequences of the hazard may have a direct or indirect effect. For instance, the direct environmental consequences of a population incident may well be associated with indirect effects which relate to the long-term health of the users.

2.3.2 Safety integrity levels

Safety integrity is a continuous quantity which is generally assessed in subjective terms as safe or unsafe, that is either one side or the other of a level which is regarded as acceptable. In practice a discrete measure is used which allows safety integrity to be expressed in quantified terms. The quantification is rather crude, typically the safety integrity of a system is expressed as one of several levels ranging from highly catastrophic hazards to a very low risk level.

The IEC in [60] postulated a mapping between these safety integrity levels and the consequences associated with the hazard concerned as shown in Fig.2.2. However, this mapping is purely illustrative. As was discussed earlier, safety integrity is affected by a number of factors other than the consequences of the hazard. These factors and their inter-relationships are illustrated in Fig.2.3.

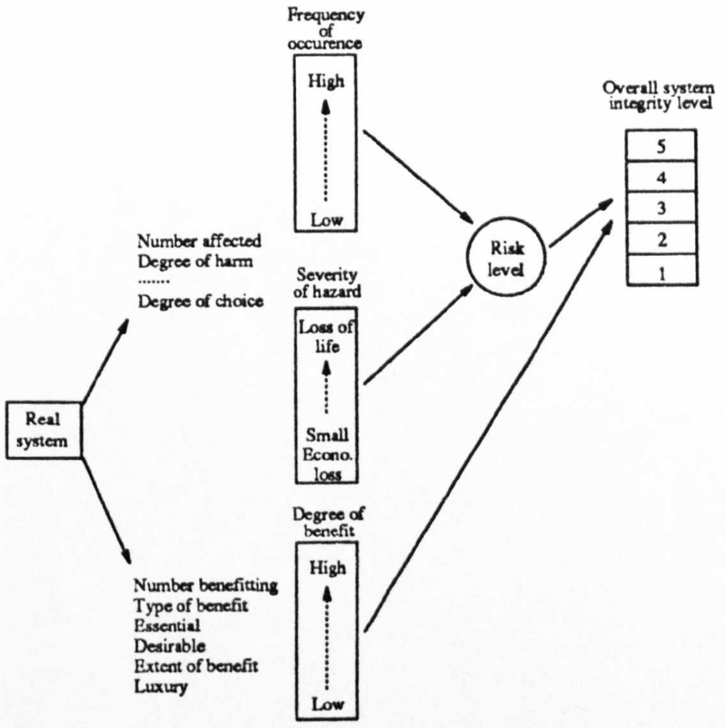


Figure 2.3: Factors affecting software integrity levels

2.4 Determination of Assessment Levels

So far, we have identified four major aspects of safety related system involving different levels,

- the severity of hazard,
- the frequency, or probability, of occurrence,
- the safety integrity required, and
- the degree of assessment required.

The level of required safety integrity relates to the means used to both develop and access the system. The level of assessment relates only to the means used to ensure that an acceptable level of safety is achieved. The severity of hazard and the frequency of occurrence, as discussed in section 2.3.1, are the basis for the selection of safety integrity and assessment levels.

Integrity level	Associated with
Low	local environmental damage
Medium	wide environmental damage
High	small scale injury
Critical	small scale loss of life
Catastrophic	wide scale loss of life

Table 2.2: Safety integrity levels

Level	Assessment
0	System overview
1	System structure analysis
2	System hazard analysis
3	Rigorous analysis
4	Formal mathematical techniques

Table 2.3: Assessment levels

When assessing safety related systems we identify five levels of hazard severity, five levels of safety integrity and five levels of safety integrity assessment, based on the work done by the HSE [60], IEC [63] and other bodies [62,66]. Table 2.2 illustrates the association we make between safety integrity level and hazard severity. Notice that the levels we use do not correspond exactly with those described by the IEC in [63]. We do not explicitly use levels of frequency of occurrence. Rather the frequency of occurrence of a hazard is implicitly incorporated into the scale of hazard severity.

Table 2.3 identifies the levels of assessment. These are associated with the levels of hazard severity and safety integrity identified in Table 2.2. The association of levels of hazard severity, frequency of occurrence, safety integrity and assessment is flexible in the sense that safety is essentially a subjective quality, based on a variety of different factors for which there are few measures and no generally agreed means of combination.

2.4.1 Level selection

One of the most difficult steps is the choice of a level at which to assess the system and its components. This is related to the level of safety integrity which the system should achieve. The required level of safety integrity summarises many interlinked points, all of which are considered,

- Potential for loss of life
- Potential for human injury
- Potential for environmental damage
- Potential for economic damage
- Potential benefit derived from the system
- Expected number of instances of the system
- Expected number of occurrences of the hazard
- Contribution of the system to safety.

The process of determining the assessment level for a particular system is represented by Fig.2.3, which summarises the relationships between the various factors which contribute to safety integrity and the contributions made by different aspects of a system to the overall safety integrity of the system.

Having defined the safety criteria and the safety integrity level which the system should achieve, the next step is to select an appropriate development technique which can ensure that the safety criteria are met.

2.5 Selection of Development Techniques

The approach is to associate with each level of assessment a number of techniques which can be used in assessing a system, as shown in Table 2.4 [60]. The most rigorous assessment level, i.e. level 4, is associated with the use of formal mathematical methods. Level 1, however, is associated with the use of a review of the

Level	Associated techniques
0	Design review
1	Checklists, Code walkthrough
2	Fault tree analysis, Event tree analysis, Cause effect graphs
3	State transition diagrams, Petri-Nets, Markov models
4	Formal mathematical specification and analysis techniques

Table 2.4: Assessment techniques

design and design procedures using checklists and a walkthrough and review of the development stages. The assessment of any system typically involves the use of techniques associated with more than one assessment level. For instance, design reviews are performed at all levels and formal hazard analysis is performed at all levels above level 2.

The concept of levels is important to the design and assessment of safety-related systems. Safety itself, and the factors which contribute to the level of integrity required and that achieved, are continuous quantities with no established metrics. The concept of levels allows some degree of measurement to be applied to these quantities and hence introduces a more concrete assessment of the system integrity.

Different techniques are associated with the different levels of assessment. These techniques model different aspects of the system. In addition to the independence of the personnel performing the assessment, diversity is therefore provided with respect to the viewpoint from which the system is assessed.

2.6 Validation and Verification

Depending on the level of assessment a variety of different verification techniques are applied, ranging from straight forward design review to formalised analyses of all or part of the system.

Different levels of safety integrity and different types of application require that differing approaches, with increasing levels of formality, are employed in system development. The validation and verification process takes this into account so that in the process of reviewing a high integrity system more stringent criteria are applied than would be the case for a low integrity system.

However, it is possible to apply a low level of verification to a system which requires a high level of safety integrity. Since the purpose of the assessment is to ensure that the required level of safety integrity has been reached, this may be sufficient. A high level of assessment will, however, provide the necessarily greater degree of assurance that the required integrity level has been achieved.

2.6.1 Informal techniques

Apart from dynamic execution testing, informal or manual methods have been the only methods available for carrying out validation and verification of the system when a fairly low level of safety integrity is required. Such informal methods include,

- walkthrough
- review
- audit/checklists
- code inspection.

As methods, however, they are by no means exhaustive, and although having the advantage of repeatibility they are not particularly effective in detecting errors introduced as a result of minor revisions. Nevertheless, such methods continue to be as cheap and effective as any in finding the majority of errors, especially when performed by experienced designers and therefore, they are most suitable for analysing systems when safety constraints are not critical.

2.6.2 Semi-formal techniques

A number of semi-formal methods are used for those levels which have a high level of associated degree of risk. Typical techniques include fault tree analysis, Markov reliability model, Petri Nets and state transition diagrams.

Just as a rigorous and complete test case analysis is equivalent to formal verification, a rigorous treatment of these analysis is equivalent to formal verification. However, it should be emphasised that a standard analysis using the above techniques applied to system design, like the testing of the system, is informal and incomplete. For instance, fault tree analysis, as it is usually performed, is an effective technique for focusing attention on some significant aspects of the design but, without the rigour of formal verification, it cannot ensure that the required level of safety is achieved.

Therefore, for safety-related systems with a very high level of associated risk, a formal treatment is essential. Although the semi-formal analysis can make significant contributions to achieving safe systems, formal specifications and verification is the only method proposed to date that can achieve in principle, if not in practice, the level of safety required for safety-related applications. Of course, formal verification should be applied in conjunction with, rather than instead of, other techniques.

2.6.3 Formal techniques

Formal verification is used to validate systems against a formal specification that defines what it is required to do. To achieve substantial improvements in system safety, techniques for validating formal specifications are important. The system verification requires that the detailed formal specifications, which is required for system verification, should themselves be verified against a simpler more abstract specification. Of course, the simpler more abstract specification may still be subject to error and must be verified against yet another simpler and still abstract specification, a process that eventually leads to a top-level specification. In the context of safety-related computing, this top-level specification should express the safety requirements of the system.

However, there is no simple systematic method for checking that a top-level specification correctly states the required safety properties of the system, since safety properties, which are included in the specification, are also specified by the user. Therefore, the specification expresses the real safety requirements of the application and must, in the end, be determined by human inspection and his/her understanding. In principle, it is the safety-related properties defined in the top-level specifications that are assured by the verification.

There are a number of tools available to assist with the formal verification of hardware and software systems. Some formal methods have, in fact, evolved from the semantic-analysis tools that were built to manipulate specifications and programs. Tools and their associated specification languages that handle subsets of first-order logic include the Boyer-Moore Theorem Prover (and the Gypsy specification language)[42], LOTOS (and the language CSP) [114], and m-EVES (with language m-Verdi) [34]. Finally, tools that handle subsets of higher order logic include HOL [49], LCF [43], and OBJ [41]. The relative advantages and drawbacks of these formal methods and languages will be discuss further in the following chapter.

2.7 Discussion

Designing for safety, particularly for computer-based systems, is an obvious goal, but up to now little guidance existed on how to achieve it. The IEC Working Group 9/10 has recently produced guidelines on design for system safety [63] and the work in this chapter are most closely related to the philosophy on which those guidelines are based and the principles underlying them. In fact, this proposed standard addresses the full lifecycle of the system development. There are two significant points to the standard from IEC WG9/10. Firstly, the standard advances a five-point scale for the levels of safety integrity so as to introduce the concept of degree of risk. Secondly, the standard does not place single emphasis on the use of formal methods for developing and assessing systems unlike the interim Def Stan 00-55 [66]. Rather it proposes the use of a package of techniques chosen from a set of such techniques such that they reflect the system integrity, the application and the industry.

However, the main difference between our approach and that proposed by the IEC standard is that in the Standard the required integrity level is selected on the basis of the level of risk associated with the system, whilst in this approach we also consider the benefits which the system contributes. Furthermore, the level of risk associated in a system as a whole is not directly related to the level of risk associated with software and other components. In fact, the overall level of safety integrity is always higher than its constituent parts.

Finally, the work presented in this chapter outlines a general framework for the development of systems based on the current state of the art. The main emphasis in this thesis as a whole is on the use of formal methods to verify the correctness, and ensure that the required level of safety integrity is indeed achieved. This chapter has laid down a methodological approach based on which formal methods are consistently introduced into, and incorporated with, the system design. The following chapter discusses the concepts of safety and risk from a mathematical point of view and illustrates how such safety objectives can be achieved in a formal development framework.

Chapter 3

Formal Specifications in Engineering

3.1 Introduction

Formal specification and verification can be used to validate systems against system requirements where the required level of safety integrity is very high, as was discussed in the previous chapter. During the verification of a system, a large number of theorems must be proved. Although these proofs are not mathematically profound, keeping track of the details of the proofs and the inter-relationships among the various theorems can be overwhelming. Furthermore, the proofs must be carried out with a very high degree of rigour to ensure the high level of safety required. The generation of a formal specification and its verification is usually carried out in a deductive system, part of which can be automated by employing a mechanical proof checker, also called an automated theorem prover. However, such a proof checker is usually far from automatic. The user must supply many trivial details of the proof, usually in a manner that must accommodate the internal working of the prover. In effect, the user must create the proof, while the verification systems is responsible for the soundness and completeness of the proof. Fortunately, experience has shown that, although mechanical theorem verifiers frequently fail to find proofs that they might be expected to find, they seldom find proofs that are invalid. The construction of proofs is not inherently difficult, but it is very time consuming.

To satisfy the mechanical theorem prover and to achieve the required level of safety, the specifications must be formal. Informal specifications in plain English may be useful in understanding the system and its documentation, but they cannot serve as a basis for verification. Even carefully written formal specifications are prone to error, and experience has shown that unverified specifications are comparable in reliability to untested programs. Indeed, the process of verification involves, in part, the correction of the formal specification.

In an attempt to obviate the inherent difficulties associated with the development of formal specification, and formal techniques in general, this chapter presents some heuristic approaches to the generation and validation of formal specifications for engineering problems. Our approach is to classify practical systems into categories and then present a formal treatment which is appropriate to each class of system.

3.1.1 System Classification

From the viewpoint of specifications, practical systems differ not only in terms of their functional properties, but also in terms of the interactions between their inputs and their outputs. We have identified, from an engineering viewpoint, two major classes of systems:

- **Memoryless systems:** whose every output is a function of the current input alone; such systems have no memory of their past inputs.

- **Memory bearing systems:** whose current output depends not only on their current input, but also on their past inputs; such systems must constantly memorise, if not an infinite history of their past inputs, at least a summary of it.

  Within memory bearing systems, we have identified an important class of systems which has important practical applications,

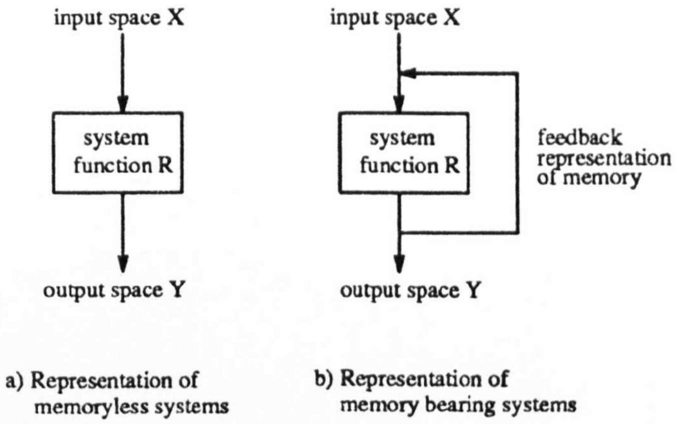

input space X
system function R
output space Y
a) Representation of memoryless systems
input space X
system function R
feedback representation of memory
output space Y
b) Representation of memory bearing systems

Figure 3.1: Representations of engineering systems

- **Continuous processes,** whose function is to compute responses to a stream of inputs. In such systems, real-time aspects have to be identified.

Hence, from the viewpoint of specifications, we distinguished between three classes of systems, memoryless systems, and continuous processes. Typical examples of the first class include a CAD package, or a theorem prover. Examples of the second class include a database management system. Examples of the third class include an operating system, a process control and monitoring system, and a database management system. However, due to the looseness of the classification criteria, a system can be classified into more than one class, as in the case of a database management system. Detailed specification examples of these types of systems are discussed later in Chapter 6 and Chapter 7. In Chapter 6, we will present a case study of a memory bearing device called Drive-by-Wire, and in Chapter 7, a detailed discussion of the specification of a CAD system is presented.

A traditional view of these types of systems is illustrated in Fig.3.1, in which the underlying principle of system modelling is based on a state machine description with or without a feedback loop. With this model in mind, in the following section, we can formally state the safety conditions and the assessment criteria to which the intended systems have to comply.

3.2 Formal Model of Safety

In state machine descriptions, a system is modelled as a single, compound function which generates a new state from an input state as shown in Fig.3.1. A formal treatment of safety is only concerned with the current behaviour of the system and therefore, it can be equally applied to machines which may or may not have memory.

Let us suppose now that the operating domain of the system under consideration can be divided into two subsets, S and U which represent the set of $safe$ and $unsafe$ states respectively. Using predicate calculus it is possible to formally state the definition of system safety such that there are three conditions to be satisfied,

1. When the current state, s_i, is contained in a set of safe states S, there is a function F that will transform the current state into the next state, s_j, which is also contained in the set of safe states S.

   $$\forall s_i \in S.\ F(s_i) = s_j \supset s_j \in S$$

2. When the current state is contained in a set of unsafe states U, there is a function F that will transform the current state to the next state, which is contained in the set of safe states S.

   $$\forall s_i \in U.\ F(s_i) = s_j \supset s_j \in S$$

3. When the current state is contained in a set of unsafe states U, and there does not exist a set of safe states for the current state to be transformed to, then there is a function F that will transform the current state to the next state with the lowest risk.

   $$\forall s_i \in U.\ \neg(\exists\, s_j \in S.\, F(s_i) = s_j \ \wedge\ \mathsf{Risk}_{(s_k)} < \mathsf{Risk}_{(s_j)}) \ \Rightarrow\ F(s_i) = s_k$$

   where S is a set of states judged to be safe and, U is a set of states judged to be unsafe, and Risk_{s_i} is the risk level associated with state s_i.

In asserting that $\text{Risk}_{(s_k)} < \text{Risk}_{(s_j)}$ consideration needs to be given to the time taken for the system to achieve state s_k, and consequently this affects the assessment of the acceptable level of risk in the design. The risk assessment is formally based on a comparison of the cumulative effects of hazards over a finite duration of time. Strategies for defining risk levels are discussed further in the following section.

3.2.1 Risk assessment

There are at least two strategies that can be adopted when considering the consequences of time in assessing a risk level. If $\text{Risk}_{(s_k)}$ is considered to be lower than $\text{Risk}_{(s_l)}$, however more time is required for the system to achieve state s_k than state s_l, a judgement can be made whether safety is best served by achieving state s_k with a low risk in a longer time than state s_l. Suppose now that state s_l has a higher risk than state s_k but a lower risk than state s_j and can be achieved in a short time, then the last safety condition, described above, can be qualified as follows,

$$\forall s_i \in U.\ \neg(\exists\, s_j \in S.\, F(s_i) = s_j \land$$
$$\text{Risk}_{(s_k)} < \text{Risk}_{(s_l)} < \text{Risk}_{(s_j)} \land T_{(s_k)} < T_{(s_l)}\,) \Rightarrow F(s_i) = s_k$$

where T_{s_j} is the time required to achieve a particular state s_j from the current state s_i. This strategy is appropriate for those instances when it is only possible to predict one state ahead.

However, if it is possible to look ahead more than one state, an alternative strategy might be to achieve a state with a higher risk for a short time in the knowledge that a state with a lower risk will be achieved ultimately. So for this strategy, safety could be expressed as the integral of the level of danger against time,

$$\text{safety} = \int P_{s_i}.\text{Risk}_{s_i}\, dt$$

where P_{s_i} is the probability of occurrence and Risk_{s_i} represents the level of risk associated with the state s_i. The value of Risk_{s_i} is subjectively judged and could influence the choice of strategy adopted due to the level of confidence in the judgement.

It is asserted in this Thesis that the definition of safety can be formally stated. However, the determination of set of *safe* states or *unsafe* states depends on a subjective judgement based on the knowledge, experience, emotion and legislation of what is acceptable at the time. For instance, in the design of a Drive-by-Wire system which is described in Chapter 6, the machine is specified as a finite state model which maps a set of legitimate input sequences to an output space.

The output space is divided into three subsets, namely, the initialisation or start-up mode, a set of safe states and a set of unsafe states, where these states are defined by appropriate conditions, for instance, an unsafe state is defined as the case when the system failed with its throttle being wide open. In this case, a number of safety strategies can be applied, such as to shutdown the system, or to switch it to manual control mode. It is then completely dependent upon a subjective judgement of the specifiers, the users and what are allowed in the safety legislation of the automotive industry to dictate the required actions. The verification of the system is observed in order to ensure that the three safety measures outlined above are indeed achieved.

However, in order to formally prove that the required level of safety is reached, the specifications must be formal. Informal specifications may be useful in understanding the system and its documentation, but they cannot serve as a basis for verification. In the following section, we describe a specification model which provides a framework for a formal development. Later in section 3.4 and section 3.5, we discuss techniques for generating and verifying such a model.

3.3 A Specification Model

A formal specification can be modelled as a tuple of 3-element represented by (X, Y, R), which includes

- A set X, called the input space of the specification. This set contains all the inputs that are likely to be submitted to the system.
- A set Y, called the output space. This set contains all the outputs that systems are likely to return.
- A relation R from X to Y, called the input-output relation of the specification. This relation contains all the pairs of the form (x, y) where $x \in X$ is a legal input sequence and $y \in Y$ is a correct output for the input sequence x. By including the pair (x, y) in R, the specification states that for input sequence x, the output y is considered to be correct. The necessary constraint for the correctness argument of a specification is that the relation R has to be *onto*, this means that there is at least an output $y \in Y$ for any input x. The uniqueness property further requires that R also needs to be *one-one*. However, in practice, there are cases where, for a given input x there may be more than one output $y \in Y$ which can be designated, and R in this case can be arbitrarily non-deterministic.

It is convenient to consider that input sequences are infinite, we represent them as follows,

$$(\ldots x_3\ x_2\ x_1\ x_0)$$

where x_0 is the current input, x_1 is the most recent previous input, x_2 is the input submitted prior to x_1, and so on. In practice, it is not necessary to look at all the input sequence in its infinity to decide whether a given output y is correct; rather it suffices to look for the subsequence of recent inputs. Specifically, there is usually an input whose purpose is to reset the system to its initial state, for instance, setting a database to empty, setting a control system to start-up, etc. and it is often adequate to consider a set of input sequences up to the most recent initialisation.

3.3.1 Model representation

Given spaces X and Y, we can subsequently define the relation R on $X \times Y$. Let $d = (W_f, A_x, R_l)$ be a system which comprises of a set of well-formed formulae W_f called the specification, and a set of axioms A_x and rules R_l. The set of specified formulae W_f has the following properties,

- they are logical formulae interpreted over the input space X and output space Y,
- they are formulae of the form $(x, y) \in R$. where x is the input sequence and y is its corresponding output.

Then the pair (x, y) is in R if and only if the formula $(x, y) \in R$ is a theorem of d. That is, the correctness of the specification W_f is guaranteed if there exists a theorem,

$$W_f \vdash \forall x. \exists y.\ (x, y) \in R$$

In fact, in this model we only consider the correctness of a specification in relation to the Customer Requirement, including the safety properties. The issue of uniqueness as discussed in the previous section has to be addressed when dealing with the safety objectives of the design. That means that R is required to be a one-one or deterministic function,

$$W_f \vdash \forall x. \exists y\, y'.\ (x, y) \in R \ \wedge\ (x, y') \in R \ \supset\ y = y'$$

When such a representation is chosen, definitional axioms are typically used to define the desired output of the specification for trivial input sequences; while inference rules define the equivalence between input sequences. The model representation is illustrated in Fig.3.2. The most natural way to define the integrity

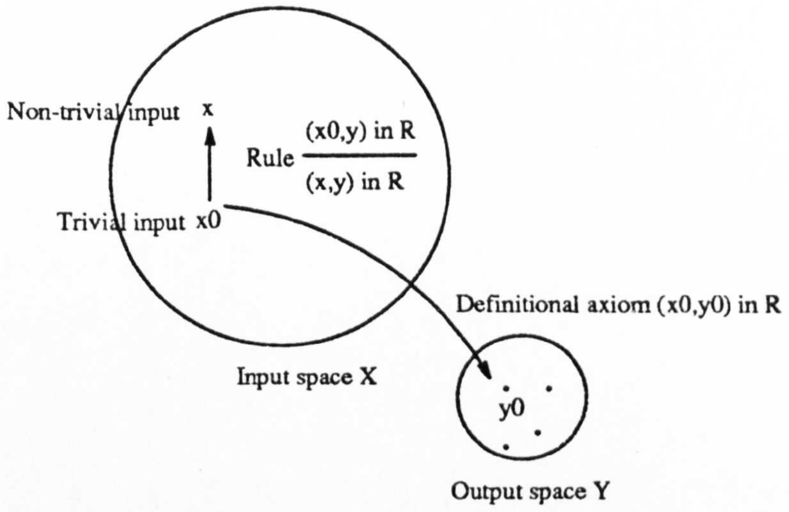

Non-trivial input x
Rule $\frac{(x0,y) \text{ in } R}{(x,y) \text{ in } R}$
Trivial input x0
Definitional axiom (x0,y0) in R
Input space X
y0
Output space Y

Figure 3.2: Specification model

of a specification is to give the necessary and sufficient condition under which it can be proved correct. This correctness condition is illustrated in the following sections.

3.3.2 Formula of correctness

Suppose now that $S_p = (X, Y, R)$ is a specification with an input space X, an output space Y and an input-output relation R. Implementations for this specification must have X as their input space, and Y as their output space. In order to match them against S_p to check for correctness, it is necessary to define a function that computes between their inputs and outputs.

The functional abstraction F of an implementation I is a function from X to Y such that,

$$F = \{(x, y) \mid I(x) = y\}$$

It follows that given a relation R and an implementation I, the function F describes what I does, while R describes what I must do. It is obvious that the correctness of I with respect to R must be expressed as a consistency condition between F and R, that is

$$I \text{ is correct with respect to } R \iff F \text{ is more defined than } R$$

In other words, F must carry more input-output information than R, i.e. the implementation I does all that R requires. More formally, the correctness statement can be formulated as a logical equivalence of the form,

$$\vdash \forall xy.\, I(x,y) \equiv F(x,y)$$

where I is a predicate representing the implementation, and F is a predicate representing the specification. For a simple system, this is often the appropriate way to state correctness and some system can only handle such equivalence [45]. However, for complex systems, it is usually wrong to require that the implementation be logically equivalent to the specification. The specification will typically be some sort of abstract description of behaviour, operating on different data types and at a different time scale to the implemented systems. In such cases, correctness is more appropriately formulated as an implication of the form,

$$\begin{aligned}&\vdash \forall xy.\, I(x,y) \supset\\ &\qquad \textbf{let } x_1 = \mathsf{Abs}_i(x) \textbf{ in}\\ &\qquad \textbf{let } y_1 = \mathsf{Abs}_o(y) \textbf{ in } F(x_1,y_1)\end{aligned}$$

where the Abs_i and Abs_o respectively maps the input and output of the implementation I, to those of the specification F.

In practice, however, it is a property of the implication that the predicate

$$\vdash \forall x.\, \mathsf{False} \supset x$$

is true for any x, i.e. *false implies anything*. Thus if the implementation $I(x,y)$ were equivalent to False then, in a mathematical sense, it would correctly implement every specification. Hence, care has to be taken when deriving correctness proofs for a particular implementation to ensure that this problem does not arise. In fact, to obviate the problem, the correctness statement can be reformulated in a more rigorous way, by combining the correctness and consistency condition as follows,

$$\vdash \forall xy. (I(x,y) \supset F(\mathsf{Abs}_i(x), \mathsf{Abs}_o(y)) \land I(x,y)$$

The second conjunct of this theorem states that for all possible input values, there are some output values which are consistent with those. If this is true then the implementation formula $I(x,y)$ is never equal to False and so the *false implies everything* problem can not arise.

3.3.3 Total and partial correctness

The specification model and its correctness statement described so far, is called a partial correctness specification. These specifications are partial because for the relation $F(x,y)$ to be true, it does not assert anything about the internal behaviour of the system. It merely means that the system described by F delivers an output y on the reception of an appropriate input x. Let us take the example of the Drive-by-Wire, presented in Chapter 6 for instance. Generally, the specification is described as a tuple of $S_p = (X, F, Y)$ where X and Y are sets of legal input and output sequences, and F is a relational representation of the system which is modelled as a finite-state machine. The partial correctness statement of the specification is asserted by a theorem of the form,

$$\vdash \forall xy. x \in X \land F(x,y) \supset y \in Y$$

More specifically, a useful theorem [1] which asserts the system behaviour in a shutdown condition can be shown in a simplified form as follows,

$$\vdash \forall s. \mathsf{SAFE}\ s_0 \land \mathsf{Nextstate}\ (s_0, s_1) \supset \mathsf{SAFE}\ s_1 \qquad (3.1)$$

where s_0 and s_1 are the input and output state vectors and the relation Nextstate is true if s_1 is an output of s_0. In this case the domain of both input and output space

[1] Detailed discussion is presented in section 6.6.3 and shown in Annex C.

is defined coherently by the predicate SAFE which in turn, defines a set of safe states of the system. What is illustrated in this example is the fact that the above theorem guarantees the partial correctness of the specification by asserting that given any sequences belonging to a suitable input space the system will deliver a legal output.

Although, theorem (3.1) is useful in determining the behavioural correctness of a specification, it says nothing about the deterministic property of the system in question. In fact, determinism is a dual problem with the termination of computer programs which can be stated formally as follows,

$$\vdash \forall x.\, x \in X \supset \exists y.\, y \in Y \wedge F(x,y) \tag{3.2}$$

Notice that this asserts that F terminates under input condition x if there is some final output state for each initial input state satisfying X. For example, the terminating condition of a finite state machine model of the Drive-by-Wire is described by a theorem of the form,

$$\vdash \forall m\ v\ p\ t.\, \exists s.\, \exists m'\ v'\ p'.\, \mathsf{Nextstate}\ (s,m,v,p)\, t = (m',v',p')$$

In fact, the correctness of the function Nextstate can be illustrated further by showing that it is also deterministic, i.e. it satisfies the relation

$$\vdash \forall x\ y_1\ y_2.\, F(x,y_1) \wedge F(x,y_2) \supset y_1 = y_2$$

In this case, the termination condition of (3.2) is sufficient and the total correctness statement of a specification can be defined by the informal equation,

$$\text{Total correctness} = \text{Termination} + \text{Partial correctness}$$

and it can be implemented by asserting, as shown in the case of the specification of the Drive-by-Wire that,

$$\vdash \forall m\ v\ p\ t.\, \exists s.\, \exists \forall m'\ v'\ p'.\, \mathsf{NextState}\ (s,m,v,p)\, t = (m',v',p')$$

In fact, the total correctness of the specification is expressed by the unique ($\exists\forall$) quantifier which states that there is one, and only one, set of output states for each occurrence of the input state s. Having defined the specification model and its correctness statements, we will discuss techniques for generating such specifications and outlines methods with which they can be validated in the following sections.

3.4 Specification Generation

Recall from Chapter 2 that the first phase of the specification process consists of expressing the user requirements in the form of a relation. This process takes place in a progressive stepwise fashion, by considering one aspect of the target specification at a time. In this section, we discuss techniques to tackle the complexity of generating a specification by breaking it down into a set of simpler subspecifications. We will discuss in turn, techniques that are applicable to memoryless systems, and then those that are applicable to memory bearing systems.

3.4.1 Memoryless systems

Let us assume that the general form of a specification of a memoryless system is a relation on some set S. Given that S is fixed, the generation of a relation R takes place in a stepwise fashion. Generally, there are three heuristic approaches which can be used in developing such specification,

- **Separate binary from unary properties**

  It is common for a specifications to be the conjunction of two kinds of properties,

  - binary properties which link input states s to output states s', and
  - unary properties which describes features of output states s' alone.

  It is good practice to distinguish between these two types of properties. More specifically, it is often more convenient, especially in the verification phase, to reduce binary properties to their simplest possible expression, thereby abstracting away from them any hint of unary properties.

Usually, a binary property has the form of an equivalent relation, and expresses some form of preservation. In contrast, the unary property serves to reduce the range of the relation, and expresses the property we wish to produce in the output. A simple example of such a distinction between binary and unary properties, which are discussed in more details in Chapter 7, can be outlined as follows,

$$R_0 = \{(b, b') \mid \mathsf{transform}\ (b, b')\} \tag{3.3}$$

$$R_1 = \{(b, b') \mid \neg \mathsf{null\_object}\ (b')\} \tag{3.4}$$

where b and b' are binary tree representations of solid objects, and the predicate transform holds if b' is actually a Boolean transformation of b, that is the predicate transform operates on two arguments b and b'. In contrast, the predicate null_object qualifies its argument b' to be a valid solid primitive if it is a non-empty object[2]. Thus, the property (3.3) is a binary relation which states that b' preserves the original binary tree b, and (3.4) is a unary relation which expresses a property of the output object b'.

- **Use only binary and unary properties**

  Some specifications can be formulated in the following terms,

  given s, find s' that maximises $f(s')$ under the constraints $p(s, s')$.

  The fact that s' which maximises function f, is not a property linking s' to s; rather it is a property linking s' to other possible occurrences of s' such that $p(s, s')$ is satisfied. As such, it does not fit well into the relational formalism, which is best suited to describing binary properties of s and s'. The key to avoiding this difficulty is to reformulate the specification in such a way that the maximality of $f(s')$ is expressed as a property of s' alone, hence fitting it

[2]Detailed descriptions of these functions are discussed in Chapter 7.

as a unary property, or as a property of s and s', hence fitting it as a binary property.

- **Stressing the right abstractions**

  A failure to find the proper abstraction in generating a specification is more often due to the lack of grasp of the requirements, on the part of the specifier than it is to some intrinsic property of the specification.

  Two broad classes of semantic abstraction functions are those that abstract preserving each system's behaviour and those that abstract preserving the system's structure,

  - Behavioural abstraction, which describes only constraints on the observable behaviour of the system. The behavioural constraint that most formal methods address is a system functionality. That is to define a relation R which maps from input space X to output space Y. Apart from functionality, other behavioural aspects, such as fault tolerance, safety, security, response time, and space efficiency can also be addressed in the behavioural abstraction.

    It is often the case that some of these aspects such as fault tolerance, are included as part of, rather than separate from, a system's functionality. If the overall correctness of the system is defined so that it must satisfy more than one behavioural constraint, a system that satisfies one but not another would be incorrect. For instance, if functionality and response time are the constraints of interest, a system producing correct outputs past deadlines would be as unacceptable as a system producing incorrect outputs on time.
  - Structural abstraction, which describes constraints on the internal composition of systems. In this thesis, we do not consider the structural abstraction of systems, rather we concentrates on the behavioural specification of both memoryless and memory bearing systems. For more detailed discussion on structural specification and other abstraction

mechanisms see, for example [49,51,79].

3.4.2 Memory bearing systems

In addition to the generation strategies outlined for memoryless systems, there are two additional strategies specifically applied for memory bearing systems.

- **The use of definitional axioms:**
  For each element of the output space Y, we generate a definition that defines for which inputs that element is to be delivered as an output. Because several inputs may share the same output, the definition will show a trivial input, and leave it to the rules to show what non-trivial inputs share the same output as the trivial one.
- **The initialisation rule:** Each memory bearing system has an operation that resets it to some predefined state that is usually interpreted as the initial state. Given such an operation, we must express in a rule, that everything that occurs before such an operation is to be ignored.

Detailed examples on the use of definitional axioms and rules for generating and reasoning about a specification are given in Chapter 6. The set of strategies presented here for the generation of the specification of the memoryless and memory bearing systems are by no means complete, they are merely indicative. In fact, there is no systematic way to check that all the requirements have been formulated completely and correctly. The only way to ensure the consistency and completeness is to perform a validation, in the form of formal reasoning, on the specification. Different approaches for system validation are discussed in the following section.

3.5 Specification Validation

In practice, when a specification is derived from a user requirement, we can never be sure that we have captured all the requirements that the user has in mind (completeness), or to have captured them faithfully (consistency). On the other hand, because the user requirements are expressed in an informal document, it is not possible to check them formally for completeness against a formal specification. The purpose of the validation step of the specification process is to check the completeness of the proposed specification with respect to the user requirements. In the following sections, we discuss in turn how the specification validation is performed for memoryless systems, then we consider memory bearing systems.

3.5.1 Memoryless systems

Just as we did in the previous section, we assume, without loss of generality, that the general form of a specification of a memoryless system is a relation with respect to some set S. The validation step raises two crucial questions that must be considered, namely,

- What form do properties take ?
- How do we check a specification against a property ?

The task of validating a specification requires the generation from the requirements of properties that the specification must meet. Properties are relations which can be stated formally as follows,

> *A specification S expresses that for input states satisfying the predicate $q(s)$, output states s' must satisfy the predicate $q'(s, s')$.*

Such a definition can in effect be captured in the relation

$$V = \{(s, s') \mid q(s) \wedge q'(s, s')\}$$

From this definition, it is implied that checking for a specification S to be valid with respect to a property V means must to validate the following conditions,

- The set of input states associated with property, V say, is included in the set of input states associated with specification S.

- Whenever applicable, the set of output states acceptable by V includes the set of output states acceptable by S.

In other words, the above validity conditions formally implies that the domain of V is a subset of the domain of S, i.e. S is more-defined than V. For instance, in Chapter 7, when solid objects are combined using pre-defined solid primitives, an important property to preserve the correctness of the operation is that it is required not only to show that the solid components are in fact satisfied some pre-defined validity conditions, but also to prove that the combined object is actually a valid solid, i.e. it also belongs to a set of homogenously three dimensional objects [3].

3.5.2 Memory bearing systems

It is obvious that the problem of validating a memory bearing specification is identical to that of validating a memoryless specification, since in both cases the specifications of memory bearing systems are hardly ever written as explicit relations, but are rather written using definitions and rules.

If specifications and properties are represented as explicit relations, then the problem of validation is the same as for memoryless systems, and will not be considered further here. Thus, suppose that specifications are represented axiomatically using explicit definitions, then in the remainder of this chapter, we will discuss specification validation for various representational forms of properties.

- **The property is a singleton**

  Consider a property of singleton that for a given input sequence x_0, the output is y_0 for a specification of the form $S = (X, Y, R)$ where X and Y are input and output space respectively, and R is an input-output relationship. This can be represented formally by the relation,

  $$V = \{(x_0, y_0)\}$$

[3] More details on solid modelling and operations are discussed in Chapter 7.

In this case, it is obvious how to validate the specification for a specific input x_0, and to check whether the specification yields the output y_0. This is exactly similar to program testing. In order to prove that a given specification meets a property of this form, we must prove the following lemmas,

$$x_0 \in X$$
$$R(x_0) = \{y_0\}$$

These lemmas are, in fact, derived from the definition of total correctness discussed in section 3.3.3. Because the first lemma is a consequence of the second, it is convenient to prove the second. However, it should be emphasized that the second lemma asserts that the application of R on x_0 results on a singleton set with y_0 as its only element. Therefore, using the principle of separation of concern, this can in turn be proved by means of the following lemma,

$$(x_0, y_0) \in R$$
$$(x_0, y) \in R \supset y = y_0$$

The first lemma ensures the partial correctness proof and the second asserts the determinism of the specification. To prove the first lemma, it is required to show that $(x_0, y_0) \in R$ is a theorem of the deductive system that represents the specification. To prove the second lemma, it is necessary to show that $y = y_0$ is a logical consequence of $(x_0, y) \in R$ in the deductive system that represents the specification R. The latter proof uses the fact that R is a possible relation that verifies the definitions of the specification.

- **The property is written as a relation**

  This is the most common form of properties expressed in any formal deductive system. Intuitively, the specification is valid with respect to the property if and only if it can be proved that the property is indeed a theorem of the formal system itself. Any subsequent rules, axioms, definitions and theorems used to derive this property, contributes to a formal proof. Detailed discussions about proof procedures and tactics are presented in Chapter 4.

3.6 Description of Formal Methods

In considering the characteristics of a formal method, it is important to distinguish between a formal system and the method of its application. A formal system is characterised by being logically self-consistent. Each expression can either be defined in terms of another, simpler expression, or in a set of axioms of the system. A formal system will also include rules for converting expressions into other, equivalent ones. In contrast, a formal method provides a framework for applying such a system to software development.

Many of the formal methods currently in use provide some or all of the following resources:

- a notation for writing expressions based on an established logical and mathematical theory,
- a language structure to allow the notation to be used for generating specification. This usually incorporates aspects such as data typing, variables and information hiding. It should be noted that in contrast to the logical system employed, the language component of a formal method may not be rigorously defined,
- a methodology for applying the above to software development; this can include guidance on specification, decomposition and proofs,
- support in the form of textbooks, courses and automated tools such as editors and theorem provers.

All the formal methods discussed in this chapter describe a system as a series of state changes. However, there is a distinction to be made between those methods which describe state change as it affects the internal data structure of the system, and those which describe state change as it effects the way the system interacts with its environment. Of these methods, we will discuss the most relevant ones, in the considered domain, in terms of their underlying logical principles. These include,

- First order logic
- Higher order logic.
- Temporal logic, and

3.6.1 First order logic

First order logic is the most traditional form of logic which deals with propositions and propositional functions, that is, those functions where the domain of the independent variable ranges over propositions. All usual logical connectives are defined as well as the existential and universal quantifier. However, interpretations of first order logic require a specified domain and symbols referring to entities within this domain. Different domains lead to different interpretations.

The language of first order logic consists of terms and formulae. However, the traditional approach in this formal system is purely logical and does not provide the common concept of function. Truth values are usually introduced by assigning them with an interpretation, this means that functions with truth values as the range, are used instead of predicates. Furthermore, since first order logic does not support proper quantifiers, formal methods using first order logic have difficulty in expressing timing in a detailed manner, and also in expressing abstraction between levels of description. Examples of first order logic systems include the Boyer-Moore theorem prover [17] which has been used to prove the correctness of a microprocessor at the register level [61]. A detailed discussion of the limitations of first order logic in relating different levels of abstraction can be found in [50,72].

However, first order logic is important both from methodological and practical point of view. In particular, studies with first order logic have revealed limitations in its expressive power and that leads to a new formalism which is based on higher order logic.

3.6.2 Higher order logic

Higher order logic represents an enhancement of the first order logic. It extends the notation of first order predicates in that the domain of variables also ranges over functions and predicates, and functions can take another function as arguments and return functions as results. For example, consider a temporal definition of Next as follows,

$$\vdash \mathsf{Next}\ p = \lambda t.\, p\,(t+1)$$

This asserts that the function Next is always true at the subsequent time point. Here p has time functions as domain, that is, function from time which is modelled as positive integers to Boolean values, and Next is a higher order function, accepting p as an argument and returning a Boolean result.

Specifications, written in higher order logic, are partial and hierarchical, and they can take account of low level timing issues. For instance, in the above example, the specification is partial since it leaves the value of Next at time zero unspecified. Furthermore, such simple elements of the specification can be defined at different levels of abstraction and consistency between levels can then be proved. A designer's knowledge of systems is structured as a set of theories, each defined by a theory presentation consisting of a set of symbol declarations and a set of axioms. Theorems can be deduced from axioms using inference rules.

Available supported tools for higher order logic include the HOL system developed by Mike Gordon and his team at the University of Cambridge. HOL as a formal system is polymorphic, i.e. it allows types containing type variables, and has Hilbert's operator to build the choice axiom which allows us to introduce new primitive formulae. In pure logic, a theory contains all possible theorems, whereas in HOL it contains only axioms and already proved theorems, and consequently the soundness and consistency of proofs are always preserved.

Higher order logic has more expressive power than first order predicates, but the trade off for this advantage involves more difficulty in performing and mechanising proofs procedures. The integration of higher order logic with functional methods is a trend for future development.

3.6.3 Temporal logic

In order to present systems practically, we must cope with a particular aspects of reality, namely time and its temporal evolution. A formal system that can reason about past and future events is called temporal logic. Temporal logic includes all usual connectives and adds some typical operators. Although there are many variants, the basic operators are,

- henceforth $\square$
- eventually $\diamond$
- next $\bigcirc$
- until U

In a temporal logic system, specifications are transferred into state diagrams using temporal logic decision procedures. This state diagram and the implementation are traversed simultaneously to look for a counter example, where the implementation does not satisfy the specification. There are two main types of temporal logic based on the way it consider time, namely linear temporal logic [57] and interval temporal logic [52].

In general, temporal logic is appropriate for specifying control systems, since it can capture time and dynamic behaviour, with clear and concise notation. In temporal logic, there are no explicit time variables; time is an implicit part of the temporal operators. As a result, the specifications of behaviour are more succinct. In addition, there is no need to express such information as an order over time variables. In higher order logic descriptions, time variables are explicit and signals are functions of time to values. Behaviour specifications have consequently become more complex. However, the advantage of embedding time variables in higher order logic systems such as HOL, is that it is not necessary to have special proof rules to deal with the temporal operators, so the system may be less complex [68,69].

Temporal logic can be embedded in higher order logic. Gordon and Hale [52] have shown that Interval Temporal Logic can be embedded in HOL. The work in Chapter 5 of this thesis, demonstrates that Linear Temporal Logic can also be embedded in HOL. There are several advantages to doing this, including the ability to mix a temporal logic description of behaviour with explicit time representation. Details on embedding temporal logic are discussed further in Chapter 5.

Embedding one logic within another mechanised logic in order to prove soundness and provide mechanical support, has been proved valuable in many cases. For instance, Hoare's logic was embedded in LCF [103], Dijisktra's weakest preconditions were embedded in HOL [8], while CSP was embedded in HOL in [26].

3.7 Discussion

It has been shown so far that formal methods always have a role in system design, and to make the verification and validation practical they have to be supported by theorem proving systems. However, to satisfy the mechanical theorem prover and to achieve the required level of safety, the specifications must be formal. Informal specifications may be useful in understanding the system and its documentation, but they cannot serve as a basis for verification. In fact, the process of verification involves, in part, the correction of the formal specification.

In an attempt to obviate difficulties associated with the development of formal specification, and formal techniques in general, this chapter presents some heuristic approaches to the generation and validation of formal specifications of engineering problems. Our approach is to classify practical systems into categories and then present formal treatments which are appropriate to each class of system. However, it should be noted that these treatments are by no means exhaustive.

Furthermore, the choice of a logical formalism for system verification has been shown to have a strong impact on the effectiveness of system development in terms of cost and efficiency. It involves a compromise between expressive power and ease of proof. A simple and restricted formalism, such as first order logic, will make the proofs of simple systems easy but may make it hard to specify complex systems simply and concisely. At the other end, a powerful formalism, such as higher order logic, makes available the results of general mathematics, allowing in principle the construction of whatever mathematical tools are needed to deal with the verification task at hand.

Higher order logic has been considered to be more rigorous than other methods, because all but very simple data types can be defined by the user. This avoids the need to use pre-defined mathematical models which may have unnecessary or undesirable properties for a particular specification. Use of a theorem prover, however, remains an activity requiring specialised knowledge and experience, and there is little practical experience to date of the application of these methods to system development. In fact, it is the motivation in this work to concentrate on the application of this particular technique to engineering problems. In the following chapters, a particular higher order logic system called HOL is described, and its use in specifying and verifying practical engineering problems will be discussed.

Chapter 4

HOL Theorem Prover

4.1 Introduction

The HOL system, developed by Michael Gordon and his team at the University of Cambridge, is a tool intended primarily for hardware specification and verification using higher-order logic. It is implemented on top of Cambridge LCF [93] and supersedes the earlier system LCF-LSM [44].

HOL is the name given to the entire theorem proving system which supports higher order logic as a formalism for writing specifications and conducting formal proofs. In cases where it is necessary to distinguish between the computer system and the species of higher order logic embedded within it, the terminology HOL system and HOL logic are used respectively.

A detailed account of both the HOL system and the HOL logic can be found in [50]. In order to make this thesis self-contained, however, a brief introduction to HOL is given in the following sections. This should enable the reader with little or no experience with HOL to follow the rest of this thesis. Some familiarity with predicate logic is assumed.

4.2 Background

The early work of Milner and others on the LCF project [43] inspired much effort in the area of mechanised theorem proving. A specialised language was developed for specifying and verifying hardware [44] which led to the development of the interim LCF-LSM theorem proving system [46]. Many examples were done using this system including the verification of a simple computer [47]. Many improvements to the LCF system have been made over the years, including an improved rewriting package [91], and a new tactics package [92]. Hanna and Daeche then independently developed the VERITAS theorem prover based on higher-order logic [53,54]. With the improved expressive power of higher-order logic becoming increasingly attractive, and with the experience gained from the LCF family of theorem provers, Gordon then developed the HOL theorem proving system [49, 50], which forms the basis of the work done in this thesis. A number of examples have been successfully completed which demonstrate the use of higher-order logic as a vehicle for specification and verification. These include the verification at the detailed timing level of a D-flipflop implemented in logic gates [53,56] for proofs done in the VERITAS and the HOL systems respectively), the verification of a ring interface chip [48], re-proof of the computer in HOL [68], and the first level proof of correctness of VIPER in the Viper research project [31].

4.3 The HOL logic

The type of higher order logic used within the HOL system is a version of Church's Simple Type Theory [29]. The HOL logic uses standard predicate logic notation in which one makes use of the propositional logic connectives denoting negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\supset$) and equivalence ($\equiv$) to connect propositions (such as properties and relations) to form more complicated sentences. Variables in such sentences are bound using universal ($\forall$) and existential ($\exists$) quantification.

Table 4.1 outlines the syntax and informal semantics of predicate logic. In the table, t, t_1 and t_2 stands for arbitrary *terms* while $t[x]$ stands for some term containing free occurrences of the variable x.

Higher order logic generalises first order predicate calculus by allowing one to

Notation	Meaning
$P(x)$	x has property P
$R(x, y)$	relation R holds between x and y
$\neg t_1$	not t_1
$t_1 \vee t_2$	t_1 or t_2
$t_1 \wedge t_2$	t_1 and t_2
$t_1 \supset t_2$	t_1 implies t_2
$t_1 = t_1$	t_1 if and only if t_2
$\forall x. t[x]$	$t[x]$ is true for all x
$\exists x. t[x]$	$t[x]$ is true for some x
$(t \Rightarrow t_1 \mid t_2)$	if t is true then t_1 else t_2

Table 4.1: Predicate logic notation

use higher order variables - i.e. variables ranging over functions and predicates. For example, the induction axiom for natural numbers can be written as,

$$\forall P. (P\,(0) \wedge (\forall n. P\,(n) \supset P\,(n+1)) \supset \forall n. P\,(n))$$

Here, the variable P is quantified and ranges over predicates; such variables are said to be higher order.

4.3.1 Terms

The HOL logic uses four kinds of terms: *variables, constants, function applications,* and *lambda expressions.*

Variables and constants are denoted by sequences of letters or digits starting with a letter. For example, x, y_1, and gnd can be names of variables or constants. The difference between variables and constants is not apparent at this stage but will be dealt with in later section on the HOL system when the notion of a *theory* is introduced.

Function applications have the form $t_1(t_2)$, where the subterm t_1 is called the *operator* and t_2 is called the *operand* (or *argument*). Due to the higher order nature of the logic, the results of function applications can themselves be functions, i.e. functions can take functions as arguments or return functions as results.

To minimise bracketing, function applications can be written as $f\ x$ instead of $f(x)$. Furthermore, application associates to the left and so, $t_1\ t_2 \ldots t_n$ abbreviates $((t_1\ t_2) \ldots t_n)$.

Lambda-expressions are the means of denoting functions within higher order logic. The term $\lambda x.t$ (where t is any expression) denotes the function f, say, defined by

$$f\ (x) = t$$

If we take t in the above λ-expression to be the expression $x + y$ such that we have the term $\lambda x.x + y$ then x is said to *bound variable*, y is a *free variable* and $x + y$ is called the *body* of the λ-expression.

4.3.2 Types

The HOL logic is a strong typed logic, i.e. all terms expressed in this version of higher order logic must have a *type*. Without types, the HOL logic would be unsound as the availability of higher order variables can give rise to a version of Russell's paradox [49]. The type system used in the HOL logic is derived from that of PPLAMBDA [43] which, in turn, descends from the type system formulated by Church [29]. It allows types to be either,

- *atomic* (e.g. *bool* to denote the sets of booleans or *num* to denotes natural numbers), or
- *compound* (i.e. those types built from atomic (or other compound) types by using *type operators*).

Examples of type operators in the HOL logic are *list*, $\longrightarrow$ and #, where list is a unary type operator used to denote a list of values (e.g. *num* list denotes a list of natural numbers) while $\longrightarrow$ and # are infixed binary type operators used to denote sets of functions and pairs respectively. For example, $(num\#num) \rightarrow bool$ denotes the type of a function with a domain of pairs of natural numbers and a range of boolean truth values.

Types in the HOL logic can contain variables. In order to demonstrate this, let us consider the function compose defined below,

$$\text{compose} = \lambda f.\, \lambda g.\, \lambda x.\, f\,(g\ x)$$

If compose is applied to two functions, f and g say, then the result would be a function which would apply g to its argument and f to that result.

For example, if not is a the boolean negation function of type $bool \rightarrow bool$ and even is a function of type $num \rightarrow bool$ which returns true if its arguments are even natural numbers and false otherwise, then the result of applying compose to not and even in that order would be $\lambda x.$not (even x) which is a function of type $num \rightarrow bool$.

On the other hand, if rnd is a function of type $real \rightarrow num$ which rounds off a positive real number [1] to the nearest natural number, and log is the arithmetic logarithmic function of type $real \rightarrow real$, the result of applying compose to rnd and then log is the function $\lambda x.$rnd (log x) of type $real \rightarrow num$. The function compose, therefore, appears to have two different types,

$$(bool \rightarrow bool) \rightarrow (num \rightarrow bool) \rightarrow (num \rightarrow bool)$$

and,

$$(real \rightarrow num) \rightarrow (real \rightarrow real) \rightarrow (real \rightarrow num)$$

Indeed it appears that it can have many different types, depending on the types of the functions f and g it is applied to. In the HOL logic, *type variables* are used to allow functions with more than one possible type to be expressed within the logic. Without type variables, a different function would have to be defined for every type because a single function is not allowed to denote several types.

[1]A real number in HOL can be modelled as 3-tuple of (*bool#num#num*) where the first and second *num* denotes the integer and decimal part respectively, and the boolean value indicates the binary sign (positive or negative) of it.

In HOL, however, it is only necessary to define a single function compose. If α, β and γ are type variables then compose is given the type,

$$(\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)$$

These type variables can be instantiated to different types according to the particular use of the function compose. Types containing type variables are called *polymorphic*.

4.3.3 Hilbert's ϵ-operator

Hilbert's choice operator, ϵ, plays a very important part in the HOL logic. It is most commonly used to denote values one knows to exist but which have no name.

More precisely, if $t[x]$ is a boolean term containing a free variable x of type α, then the term $\epsilon x.\, t[x]$ denotes some value of type α, a say, such that $t[a]$ is true. For example, $\epsilon.(7 < x) \wedge (x < 9)$ denotes 8 while the term $\epsilon x.\, x \geq 0$ denotes some unspecified positive number.

In the case where there is no value a such that $t[a]$ is true, then $\epsilon x.\, t[x]$ denotes a fixed but unspecified value of type α. For example, $\epsilon n : num. \neg(n = n)$ denotes an unspecified number. The notation *term:type* is used within the HOL logic to explicitly specify the type of a term. No further detail regarding ϵ is given here. For a thorough discussion of Hilbert's ϵ-operator see [73]. A detailed description of how Hilbert terms are included in HOL to build in the Axiom of Choice [55] is given in [49].

The features of the HOL logic which are necessary to facilitate the understanding of the rest of this thesis have now been covered and we now go on to show (very briefly) how the logic is implemented in the HOL system. First, however, a brief introduction to ML, the meta-language embedded in the HOL system and with which most of the system is coded, is presented.

4.4 The Meta-Language

The aim of this section is to give an introduction to the ML language; mainly covering those features which are discussed later on in this thesis. The version of ML described here as part of the HOL system is not Standard ML [115] but the LCF meta-language version described in the ML Handbook [33], where a complete description of the ML syntax and semantics is presented.

Recursive functions can be defined in ML in almost the same ways as ordinary functions. For example, the factorial function, fact, can be defined recursively as,

$$\text{letrec fac } n = \text{if } n = 0 \text{ then } 1 \text{ else } n * (\text{fact } (n + 1))$$

Of course, fact can be defined iteratively but since we will not be using iterative loops anywhere in this thesis, the reader is referred to [33] for further details.

One can also represent functions in ML as lambda-expressions. The following two definitions of a function f are equivalent,

$$(\text{let } f\ x = e) \equiv (\text{let } f = \lambda x.\, e)$$

Lists in ML are represented by a sequence of objects separated by semi-colon and enclosed within square brackets. All objects within a list must be of the same type. The expressions [], [1;2;3] and [true;false] are all examples of lists. A list such as [1;true] will not type check as the objects in the list, 1 and true, are of different types. The standard list functions are,

- hd - returns the head of a list (e.g. hd[1;2]=1),
- tl - returns the tail of a list (e.g. tl[1;2]=[2]),
- null - boolean function which checks if a list is empty,
- · - infix operator (e.g. 1 · [2] = [1;2]),
- @ - infix append operator (e.g. [1] @ [2]=[1;2]).

Another data type represented in ML is string. Strings are any sequence of characters enclosed within single quotes (e.g. 'This is a string'). There are also standard functions on strings such as concat, explode and implode.

Variables and functions in ML can be polymorphic. The notion of polymorphism has already been explained in section 4.3.2 and so it will suffice here to give an example of how a polymorphic function is defined in ML. Consider the function map defined as follows,

letrec map $f\ l$ = (null l) $\Rightarrow$ [] | f (hd l).(map f(tl l))

Since the type s of the two arguments f and l are not specified in the definition of map, ML assumes the types of f and l to be polymorphic and defines map to be a polymorphic function of the type,

$$(* \rightarrow **) \rightarrow (*)list \rightarrow (**)list$$

Sequences of asterisks are used to denote type variables in ML. The difference between ML and logic types are briefly explained in the next section.

There are three ways in which one can define new types in ML. The first is by using the command new_type_abbrev to define new names to abbreviate previously defined types. This does not really define a new type but simply binds a name to a previously defined type. This is useful for shortening long type names or for renaming types more appropriately. For example,

new_type_abbrev *intpair* = *int#int*

defines a type *intpair* to denote pairs of integers.

The two other ways of defining are used to define altogether new types rather than just abbreviations. New types can be defined to be *concrete* or *abstract*.

Concrete types are used when the objects in the type can be enumerated into subgroups. For example, the declaration,

type $signal = hi \mid lo \mid float$

defines a new type signal which has three possible values hi, lo, or $float$.

The other way of defining types is by using abstraction. The ML command abstype allows one to make an abstract type declaration. While defining a type ty, say, there are two primitive functions, abs_ty and rep_ty, which are usable within the context of type definition. The function abs_ty maps the representation of ty to ty while the function rep_ty maps ty to its representation. One can also define, within the type declaration itself, further primitive functions to manipulate the new type.

Consider, for example, the type declaration below which introduces a new type *trigger*, represented by the type *bool*.

```
abstype trigger = bool
    with on = abs_trigger true
    and off = abs_trigger false
    and bool_of = rep_trigger
    and inv c = abs_trigger (¬(rep_trigger c))
    and clock n = λt.(n = 0) ⇒ (abs_trigger false) |
                  ((t/n) * n = t) ⇒ (abs_trigger true) | (abs_trigger false)
```

The type declaration makes use of the two locally available functions abs_trigger and rep_trigger to define the following primitive functions,

- on and off - two constants of type trigger represented by true and false respectively.
- bool_of - a function of type $trigger \rightarrow bool$ which maps a value of type trigger to its representative of type bool. For example, bool_of on = true.
- inv - a function of type $trigger \rightarrow trigger$ which inverts values of type

trigger, i.e. (inv $on = off$) and (inv $off = on$).

- clock - a function of type $num \rightarrow num \rightarrow trigger$. The application (clock n) returns a function which models a clock with a pulse interval of n. Thus when n is 0, clock (0) returns a function which models a clock which is always off (pulse interval is 0, i.e. no pulse). The application clock (1), on the other hand, returns a clock which is always on (pulse interval is 1, i.e. constant pulse). The application clock (2) models a clock which toggles on and off alternately, and so on. Hence, in the application (clock n t), n determines the frequency of the clock and t represents the time at which the clock value may be evaluated.

4.5 The HOL System

Terms in HOL logic are represented in ML by enclosing them in double quotes,. The syntax of HOL terms has been described in section 4.3.2 although in practice, various combinations of ascii characters are used to represent those logic symbols not supported by the ordinary ascii terminal. For example,

$$\forall a\, b. a \supset b \equiv \neg a \vee b$$

is represented by

```
!a b. a ==> b = ~a \/ b
```

In the rest of this thesis we shall use the notation of section 4.3.2 (i.e. we will be using $\forall$ instead of `!` and $\supset$ instead of `==>`). For a detailed account of the representation of the logic in the HOL system see [50].

When a HOL term is entered in ML, it is type-checked (according to the type rules in logic - not ML) and, if successful it is given the ML type *term*. Care should be taken here not to confuse the terminology HOL terms and ML expressions and, moreover, HOL types and ML types. The rule is as follows,

- A HOL term is a special kind of ML expression and is distinguished by a pair of double quotes enclosing the logical term. HOL terms have an ML type called *term*.
- A HOL type is the type of a HOL term and forms an ML type called *type*. HOL types are expressions of the form ":...".

For example,

- (1,2) is an ML expression with type (*int#int*).
- "(1,2)" is an ML expression with type *term* (since anything enclosed within double quotes represents a HOL term). The HOL type of this term is (*num#num*).
- Likewise, ("1","2") is an ML expression with type (*term#term*) where each term has HOL type *num*.
- ":*num*" is an ML expression with ML type *type*. It represents the HOL type *num*.

4.5.1 Theories: definitions, axioms and theorems

In [49], a theorem is defined as a sequence that is either an axiom or follows from other theorems by *rules of inference*, where

- a *sequent* is a pair (Γ,t) consisting of a finite set of boolean terms Γ (called *assumptions*) and a boolean term t (called a *conclusion*),
- an *axiom* is a sequent postulated to be a theorem, and
- *rules of inference* are procedures for deducing new theorems from existing ones (see section 4.5.2).

When a sequent (Γ,t) is a theorem it is written as $\Gamma \vdash t$ or, if Γ is empty, as $\vdash t$. Certain types of axioms are classed as *definitions*. Definitions are axioms of the form $\vdash c = t$ where c is a constant not previously defined and t is a term containing no free variables. Of course, this kind of axiom is always safe as it merely defines an abbreviation. Ideally, all axioms should be of definitional form since the freedom to postulate arbitrary axioms allows the introduction of inconsistencies.

To make a definition, prove a theorem, or declare a new HOL type, one must first enter a *theory*. A theory is a collection of types, type operators, constants, definitions, axioms and theorems.

New constants and types can be declared within theories. The distinction between variables and constants is, therefore, that variables are those terms (excluding function applications and λ-expressions) which are not defined as constants within a theory.

Theories can have other theories as parents. If one is working within a theory, th say, and an object from theory th' is required in th, then th' must be declared a parent of th. If th' is a parent of th then all types, constants, definitions, axioms and theorems available in th' are available in th. Thus, th is said to be a *descendant* of th'.

4.5.2 Primitive inference rules

Theorems in HOL are presented by values of type *thm* and must be distinguished from values of type *(term list)#term*. To obtain a new value of type *thm*, one must apply a sequence of events (constituting a *proof*) to either axioms or previously proved theorems.

Such procedures for deriving new theorems are called *rules of inference*. The following is an example of a rule of inference called *modus ponens*. The example uses standard natural deduction notation where t_1 and t_2 denote arbitrary terms, the theorems above the horizontal line are called the *hypotheses* of the rule and the theorem below the line is called the *consequent*.

$$\frac{\Gamma_1 \vdash t_1 \supset t_2 \qquad \Gamma_2 \vdash t_1}{\Gamma_1 \cup \Gamma_2 \vdash t_2}$$

Hence, if we have a theorem of the form $\Gamma_1 \vdash t_1 \supset t_2$, say

$$y > 1 \vdash y \geq y \supset y^2 > y$$

and we also have the theorem which says that

$$\vdash y \geq y$$

it means that the antecedent of the implication in the first theorem is true, $\Gamma_2 \vdash t_1$, then by the rule of modus ponens the theorem

$$y > 1 \vdash y^2 > y$$

is derived. It should be noted that in the example above, the assumption Γ_2 is empty.

Inference rules are represented in the HOL systems as functions in ML. The core of the HOL system is made up of a small set of inference rules called *primitive inference rules* and a small number of definitions and axioms from which all the standard rules of logic can be derived. Indeed, one can derive further inference rules (called *derived inference rules*) which can be justified solely on the basis of these primitive inference rules and axioms.

The choice of primitive inference rules and primitive axioms in HOL is, to a certain extent, arbitrary, although it is desirable to keep them as small in number as possible so that the implementation of the logic can be kept simple and clean.

For the purpose of this thesis, it is not important to list all the inference rules and axioms. The reader is referred to [49] for further details including a complete lists of axioms and inference rules in the current version of HOL.

4.5.3 Tactics and tacticals

In the previous section we described rules of inference and how they can be used to carry out a proof. One starts with a set of definitions and theorems and manipulates them using the inference rules until the desired theorem to be proved is achieved. In other words, truth is preserved from truth. This form of proof is sometimes called *forward* proof.

The HOL system supports another way of carrying out a proof called *goal directed proof* or *backward* proof. The problem with forward proofs is that it can often be difficult to foresee which definitions and theorems are required to prove the end result, especially if the proof is long and complicated. The idea of goal directed proof is to do the proof backwards, i.e. start from the desired result (called the *goal*) and manipulate it until it is reduces to a subgoal which is obviously true.

A *tactic* is an ML function which reduces goals to subgoals. The concept of tactics was invented by Robin Milner [43]. They are used for goal directed proving as described above. Tactics are written in a similar notation to inference rules but with a double horizontal line. For example, mathematical induction can be coded as tactic of the form,

$$\frac{\forall n.\, P[n]}{\overline{P[0] \qquad \forall n.\, P[n] \supset P[n+1]}}$$

If the induction tactic is applied to a term of the form $\forall n.\, P[n]$, then the two subgoals $P[0]$ and $\forall n.\, P[n] \supset P[n+1]$ are generated.

A goal consists of a pair of values and has ML type (*term list#term*). The first element of the pair denotes the assumption list and the second element is the term to be proved. A theorem is proved by applying tactics to every subgoal generated until all subgoals are shown to be true, without the addition of invalid assumptions [2].

Tactics can be combined together by using certain ML functions called *tacticals*.

[2]For a more detailed discussions on goal directed proofs see, for example [50]

An example of a tactical is THEN where, if t_1 and t_2 are tactics then t_1 THEN t_2 evaluates to a tactic which first applies t_1 to a goal and then applies t_2 to the resulting subgoal or subgoals.

4.6 Summary

The LCF approach to theorem proving used in HOL ensures the soundness of any proof done in the system. This approach, however, is computationally very expensive. Completely formal proofs of even simple theorems in higher order logic can take many primitive inferences, and when these proofs are done in the HOL system, all the inferences involved must actually be carried out by executing the corresponding ML procedures.

There are, however, two important features of the HOL system which, together, allow efficient proof strategies to be programmed. The first of these is a feature inherited from LCF: theorems proved in HOL can be saved on disk and therefore do not have to be generated each time they are needed in future proofs. The second feature is the expressive power of higher order logic itself, which allows useful and very general lemmas to be stated in the logic. The amount of inference that a programmed proof rule must do can therefore be reduced by pre-proving general theorems from which the desired results follow by a relatively small amount of deduction. These theorems can then be saved and used by the derived inference rule in future proofs. This strategy of replacing run-time inference by pre-proved theorems is possible in HOL because type polymorphism and higher-order variables make the logic expressive enough to yield theorems of sufficient generality.

In the following chapters we move on to show how HOL can be applied to specifying and verifying the functional behaviour of systems. However, the appropriate HOL assertions in the subsequent discussions will be presented in short-hand notations. That is only definitions and derived theorems are included, fuller inference procedures and proof tactics are included in the appropriate annexes. This chapter, although not intended as a HOL manual, has served as an introduction to the main features of HOL which will facilitate a thorough understanding of the rest of this thesis. Further information on all aspects of the HOL theorem proving system can be found in the various references suggested throughout the text (for example [49,50]).

Chapter 5

Embedded Temporal Logic

5.1 Timing Verification

In a control system, suitable controller characteristics such as latency or task scheduling are vital means of satisfying the required system performance. These static behavioural relationships can be easily specified by a suitable model using, for instance, finite-automata theory [77]. However, in order to verify control systems, such as the Drive-By-Wire system, described in Chapter 6, which is operated in real-time, it is necessary to show that the internal dynamics of the controller satisfies the required safety and timing characteristics. Therefore, it is necessary to include a mechanism introducing timing constraints into the specification so that dynamic behaviour can be reasoned about.

There are a number of formal techniques which support the specification and verification of the temporal behaviour of systems. The following section outlines these techniques and discuss their relative merits compared to our approach of using higher-order logic.

5.1.1 First order logic

Hunt [61] uses the Boyer-Moore theorem prover to prove the correctness of a 16-bit microprocessor implemented at the register-transfer level. The behaviour of combinational circuits is described by recursively defined functions. Sequential circuits are modelled using functions with explicit clock arguments. These functions call themselves recursively once each clock tick. Thus time is discrete and behaviour at any given level is related to the representation of a single clock. Since the logic is first-order and there are no quantifiers, Hunt has difficulty formally relating clocks at different levels of specification. He overcomes this difficulty by using *oracle* functions which guess the exact number of time steps which a low level representation needs to execute to be equivalent to the next higher level specification. If higher order logic is used as the specification language, abstraction between levels can be expressed directly, and such oracles are not required [31].

5.1.2 Temporal logic

Another well-known formalism for expressing timing behaviour is temporal logic. An early application of temporal logic to hardware verification is the correctness proof for the design of an arbiter given by Bochmann [15]. Bochmann uses temporal operators such as $\Box$ (henceforth) and ∇ (eventually) to capture the time-dependent properties of hardware components. The arbiter correctness proof in [15] is done by proving that a collection of 'invariant assertions' hold for all states that can be reached from the initial state of the device. The correctness of the design then follows from these invariant assertions.

5.1.3 Interval temporal logic

Moszkowski [88] defines a logical formalism for specifying hardware called ITL (Interval Temporal Logic). The truth of a formula in ITL is defined relative to a finite interval of discrete time. Modal operators are used to express temporal concepts in terms of these time intervals. Moszkowski shows how this can be used to describe formally a wide variety of time-dependent aspects of hardware behaviour. Some extensions to ITL and further applications of ITL to hardware verification are discussed by Leeser [72] and Hale [52].

5.1.4 Other techniques

Other temporal formalisms to express real-time constructs include Timed Petri nets [74] which are an extension to Petri nets, and Timed-stability [96], CSP-R [70]which are extensions of CSP [58].

5.1.5 Higher order logic

It is obvious that temporal logic is useful for specifying and reasoning about relative order set of events. In temporal logic there are no explicit time variables; time is an implicit part of the temporal operators. In higher order logic descriptions, time variables are explicit and signals are functions from time to values, hence timing behaviour specifications are frequently more complex. However, the advantage of using higher-order logic is that it is not necessary to have special proof rules to deal with temporal logic operators, so the proof system is less complex.

Temporal logic can be embedded in higher-order logic; for example Gordon and Hale [52] have shown how ITL can be embedded in HOL. Several advantages result from doing this, including the ability to mix a temporal logic description of behaviour with explicit time representations.

Our solution to this problem is to embed a form of temporal logic, called Linear Temporal Logic [39,64] in higher-order logic by regarding a set of temporal operators as abbreviations for higher-order functions. This approach results in succinct specifications and simplifies the verification task by hiding some low-level aspects of proof, in particular, the use of mathematical induction, in derived inference rules for temporal logic operators.

5.2 Temporal Operators

Temporal logic are often treated as primitive systems based on semantic definitions for temporal logic operators. However, the higher-order logic framework allows us to regard temporal logic operators as simply abbreviations for higher-order functions. The following definitions yield temporal logic operators for a form of temporal logic known as Linear Temporal Logic [64].

5.2.1 Basic operations

A number of equivalent functions in predicate calculus can be expressed as temporal constructs using explicit time variables,

$$\vdash \forall x\ y.\ x \ \mathsf{And}\ y = \lambda t.\ x\ t \wedge y\ t$$

$$\vdash \forall x\ y.\ x \ \mathsf{Or}\ y = \lambda t.\ x\ t \vee y\ t$$

$$\vdash \forall x.\ \mathsf{Not}\ x = \lambda t.\ \neg x\ t$$

$$\vdash \forall x\ y.\ x \longrightarrow y = \lambda t.\ x\ t \supset y\ t$$

For simplicity, we use two basic operators, Next and Until, from which we can define many other useful operators including, interrupts and timeouts.

The function Next p is true on an interval if p evaluates to true on the successive intervals,

$$\vdash \forall p.\ \mathsf{Next}\ p = \lambda t.\ p\,(\mathsf{SUC}\ t)$$

The function Until is an infix operator which takes two predicates p and q and it evaluates to true if q holds at some point in time and p holds at least until then,

$$\vdash \forall p\ q.\ p\ \mathsf{Until}\ q = \lambda t''.\ \exists t.\ q\ t \wedge \forall t'.\ t' < t \supset p\ t'$$

The function Sometime p holds if p is eventually true in some state. It should be noted that the Boolean values True and False are temporal equivalences of the usual first-order values, ie. it has a HOL type of *num* → *bool*,

$$\vdash \forall p.\ \mathsf{Sometime}\ p = \lambda t'.\ \exists t.\ t > t' \wedge p\ t$$

$$\vdash \mathsf{True} = \lambda t.\ \mathsf{T}$$

$$\vdash \mathsf{False} = \lambda t.\ \mathsf{F}$$

$$\vdash \mathsf{Arb} = \lambda t.\ (\mathsf{ARB} : *)$$

and, Arb specifies an untyped variable which has an arbitrary value at time t. The Arb function is useful when defining don't care conditions in digital logic.

The dual construct, Always p, holds if the formula p becomes true on all subsequent intervals; in other words, there is no such state in which p does not hold.

$$\vdash \forall p.\, \mathsf{Always}\ p = \lambda t'.\forall t.\, p\ t$$

It can easily be proved that,

$$\vdash \forall p.\, \mathsf{Sometime}\ p = \mathsf{True\ Until}\ p$$

$$\vdash \forall p.\, \mathsf{Always}\ p = \mathsf{Not}\ (\mathsf{Sometime\ Not}\ p)$$

The function Eq is a temporal infix operator for relating a signal to a particular value in terms of equality,

$$\vdash \forall f\ c.\, f\ \mathsf{Eq}\ c = \lambda t.\, f\ t = c$$

5.2.2 Timing conditions

The function First evaluates to true if p holds only at time t and does not become true at any instance before then,

$$\vdash \forall p\ t.\, \mathsf{First}\ p\ t = p\ t\ \wedge\ \forall t'.\ t'\ <\ t\ \supset\ \neg p\ t'$$

The formula Last p is true if p is true on, and only on, the final state. In other words, the computation terminates as soon as the formula p becomes true. Related functions are Fin and Keep. The formula Fin p is true if p is true on the final state, and Keep p is true if p is true on every state except the last.

$$\vdash \forall p\ t_1\ t_2.\, \mathsf{Last}\ p\ (t_1, t_2) = (\forall t.\, t_1 < t \wedge t < t_2 \supset \neg p\ t) \wedge p\ t_2$$

$$\vdash \forall p\ t_1\ t_2.\, \mathsf{Fin}\ p\ (t_1, t_2) = t_1 < t_2 \wedge p\ t_2$$

$$\vdash \forall p\ t_1\ t_2.\, \mathsf{Keep}\ p\ (t_1, t_2) = t_1 < t_2 \wedge (\forall t.\, t_1 < t \wedge t < t_2 \supset p\ t)$$

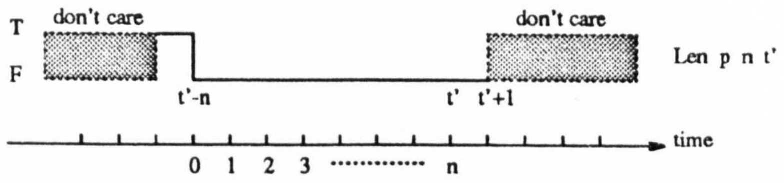


Figure 5.1: Representation of Len

Thus, it can easily be proved that,

$$\vdash \forall t_1\ t_2.\, t_1 < t_2 \wedge \mathsf{Last}\ p\,(t_1, t_2) = \mathsf{Keep}\ (\mathsf{Not}\ p)\ (t_1, t_2)\ \wedge\ \mathsf{Fin}\ p\,(t_1, t_2)$$

For example, Last $(A = 1)$ asserts that the computation terminates as soon as $A = 1$ becomes true.

5.2.3 Defining time markers

It is useful to define a marker from which elapsed time can be calculated. The function Length $p\ n$ evaluates to true when the event p has occurred for at least n units of time before the time instant t. It means that the relation p only holds at time t and does not holds on subsequent intervals. More formally, the function satisfies the inductive property below,

$$\vdash \mathsf{Length}\ p\ 0 = \lambda t.\, p\ t\ \wedge$$
$$\mathsf{Length}\ p\ (\mathsf{SUC}\ n) = \lambda t.\, \mathsf{Length}\ p\ n\ t \wedge \neg p\,(\mathsf{SUC}\ (n + t))$$

where SUC n is a function which returns the successor of n, and the function Len $p\ n$ is then defined as,

$$\vdash \forall p\ n.\, \mathsf{Len}\ p\ n = \lambda t'.\, \mathsf{Length}\ p\ n\ (t' - n)$$

It means that, Len evaluates to true at a particular instant of time t' if the event p has occurred at sometime n in the past, and it has not happened since then [1]. A timing diagram of the Len function is illustrated in Fig.5.1.

[1]Function Len is not concerned with value of p just before time $t' - n$ and after time t'.

Each operator, described above, is defined in terms of a function of the type *num* → *bool*, which maps discrete points in time, modelled by the natural numbers, to Boolean values. These operators can be combined with variables such as x and y to form assertions in temporal logic. In both first-order logic and higher-order logic, every assertion, for instance, the relation

$$\forall b.\ b \lor \neg b$$

is a Boolean expression which is either true or false [2]. However, an assertion in temporal logic such as,

$$x \longrightarrow \text{Sometime}\,(y\ \text{Until}\ z)$$

is only true or false relative to a particular instant in time. When stated as an assertion, this is informally understood to mean that the assertion is true at every instant in time. However, to formally represent temporal logic assertions as assertions in higher-order logic framework, we introduce a notion of validity where a temporal assertion is valid if and only if it is true infinitely often.

$$\vdash \forall p.\ \text{Valid}\ p = \forall t.\ p\ t'$$

Furthermore, if p is a time-dependent relation then it is an increasing function, that is, it satisfies the predicate,

$$\begin{aligned}
&\vdash \forall t_1\ t_2\ t_3\ t_4.\ \text{Last}\ p\,(t_1,t_2) \land \text{Last}\ p\,(t_3,t_4) \land (t_1 = t_3) \supset (t_2 = t_4) \land \\
&\quad \forall n_1\ n_2.\ n_1 < n_2 \supset (p\ \text{Proj}\ n_1) < (p\ \text{Proj}\ n_2) \land \\
&\quad \forall n.\ \text{Last}\ p\,(p\ \text{Proj}\ n, p\ \text{Proj}\,(\text{SUC}\ n))
\end{aligned}$$

The first theorem asserts that the time interval during which predicate p holds is unique. The second theorem asserts the well-ordering properties of natural numbers, that is, the succesive mapping of p from an abstract time scale to a

[2]It is true in this case; the law of Excluded Middle

concrete time scale is always in increasing order, and the last theorem states that there is always a successive time interval of n at which p can be projected. The validity condition can then be specified as follows,

$$\vdash \forall p.\ \mathsf{Valid}\ p = \forall t.\ p\ t'$$

Powerful inference rules for direct manipulation of temporal logic assertions can be derived in higher-order logic from the above definitions. Many such rules effectively simplify what would otherwise be tedious and repetitive patterns of inference. For example, the following theorem provides a particularly useful rule; this rule achieves, in a single step, an inference which would otherwise involve a proof by mathematical induction.

$$\vdash \forall p\ q\ r.\ \mathsf{Valid}((\mathsf{Always}\ p\ \mathsf{And}\ (q\ \mathsf{Until}\ r)) \longrightarrow ((q\ \mathsf{And}\ q)\ \mathsf{Until}\ (p\ \mathsf{And}\ r)))$$

A list of definitions and pre-proved temporal logic theorems is included in Annex B. A recent survey article [27] describes traditional logic (which includes higher-order logic) and temporal logic as two alternative kinds of pure formalisms for reasoning about time-related problems. But when temporal logic operators are simply abbreviations for higher-order functions, anything which can be done with temporal logic operators can also be done without them using explicit time variables. In fact, we have taken a mixed-mode approach of using both temporal operators and explicit time variables. The right mixture of temporal operators and explicit time variables yields relatively succinct specifications and much easier proofs.

Previous work by Hale [52] demonstrated that the idea of embedding temporal logic in higher-order logic was of practical use. Leeser [72] has also embedded temporal logic in a theorem proving environment to reason about hardware. Both Hale and Leeser used another form of temporal logic called Interval Temporal Logic developed by Moszkowski [88,89] to reason about digital systems in general. However, the more ordinary form of temporal logic captured in our

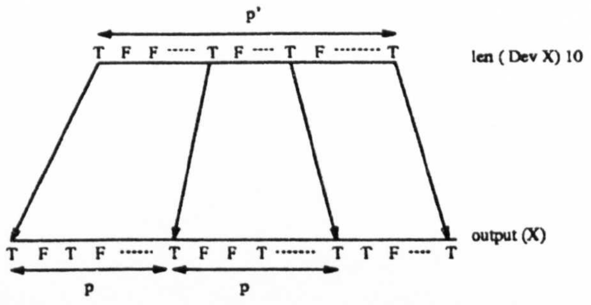

p'
T F F ····· T F ···· T F ········ T
len (Dev X) 10
output (X)
T F T F ······ T F F T ······· T T F ···· T
p
p

Figure 5.2: The projection p When p'

definitions above is adequate for the very specific purpose of reasoning about the design of control systems.

The following sections discuss some ideas and operations that are particularly relevant to the description of real-time systems. The first of these new operations is the temporal projection, which may be used for the interruption and later resumption of a process. Another important idea in real-time programming is exception handling, and a general mechanism is defined for this.

5.3 Temporal Projection

Temporal projection is one way to model a system on a number of different timescales. For example, it may be that one needs to monitor the behaviour of some device, $dev(X)$ say, but not all the time. Suppose, in fact, that the value of X is to be output on every tenth state. This is most easily achieved using the projection operation When,

$$\mathsf{Always}\ (output\ (X)\ \mathsf{When}\ (\mathsf{Len}\ dev\ (X)\ 10))\ t$$

The projection of Len $dev\ (X)$ 10 onto $output\ (X)$ repeatedly interrupts the output process by inserting an interval of length 10 between each pair of states on which X is output.

The projection p When p' holds if there is a selection of time points on which p' is true and such that p is true on the subinterval between each pair of adjacent points in the selection. This mapping process is illustrated in Fig.5.2 above.

Suppose that p represents a real-time clock which ticks every second then p is true at every instant of time. Suppose also that p' is another process which occurs in parallel, then the problem of mapping between processes, p' onto p, reduces to the mapping of p' alone from an abstract time scale to a concrete time scale. The abstract time scale is the state representation on which p' is true infinitely often.

5.3.1 Mapping function

To derive a definition for When, the first step is to define a general purpose function Mapped which uses a predicate p to construct a mapping f from an abstract time scale to a concrete time scale. The following definition of Mapped is similar to the TimeOf definitions given in previous work by Dhingra [37], Herbert [57] and Melham [79]. It is defined in terms of the predicates First and Last using primitive recursion,

$$\begin{aligned} &\vdash \mathsf{Mapped}\ p\ 0 = \lambda t.\ \mathsf{First}\ p\ t\ \wedge \\ &\quad \mathsf{Mapped}\ p\ (\mathsf{SUC}\ n) = \lambda t.\ \exists t'.\ \mathsf{Mapped}\ p\ n\ t'\ \wedge\ \mathsf{Last}\ p\ (t', t) \end{aligned}$$

The first point on the abstract time scale corresponds to the first time that the predicate p is true with respect to the concrete time scale. Subsequent points on the abstract time scale are defined recursively. The next point after n on the abstract time scale, ie. $n+1$, corresponds to the next time that p holds with respect to the concrete time scale, as shown in Fig.5.3.

This primitive recursive definition captures the idea that p is true for the n^{th} time at time t. However, there is no guarantee that such time t exist for all values of p and n.

In order to use this definition, it is necessary to show that if p is true infinitely often then for all n there is a unique time t at which p is true for the n^{th} time.

The predicate Valid can be used to express the fact that if p is true infinitely

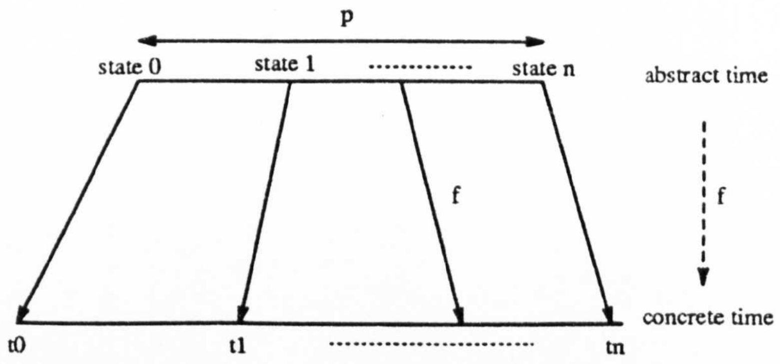


Figure 5.3: Mapping from abstract states to time

often then the time at which p is true for the n^{th} time exists for all n, that is,

$$\vdash \forall p.\,\mathsf{Valid}\ p \supset \forall n.\,\exists t.\,\mathsf{Mapped}\ p\ n\ t$$

It is also the case that Mapped defines a unique time t for each value of n. Formally,

$$\vdash \forall p\ n\ t\ t'.\ \mathsf{Valid}\ p\ \supset\ \mathsf{Mapped}\ p\ n\ t\ \wedge\ \mathsf{Mapped}\ p\ n\ t'\ \supset\ (t = t')$$

The above theorems can be combined to prove the uniqueness of the time mapping process as follows,

$$\vdash \forall p.\,\mathsf{Valid}\ p\ \supset\ \forall n.\,\exists ! t.\,\mathsf{Mapped}\ p\ n\ t$$

This means that Mapped is a one-one function which corresponds to a unique mapping between two sets of concrete and abstract time points, i.e. it satisfies the relation,

$$\vdash \forall f.\,\mathsf{ONE\_ONE}\ f = \forall x_1\ x_2.\,(f\ x_1 = f\ x_2) \supset (x1 = x2)$$

Given these theorems, the relation Mapped can now be used to define the function Proj. Using the Hilbert ϵ-operator, the definition of Proj is given by the following equation,

$$\vdash \forall p\ n.\,p\ \mathsf{Proj}\ n = \varepsilon t.\,\mathsf{Mapped}\ p\ n\ t$$

This means, p Proj n denotes some time t such that Mapped p n t is true or, if no such time exists, p $\mathsf{Proj} n$ denotes an arbitrary number. With the use of the ϵ operator, this definition makes the term p Proj n always denote a total function; p Proj n is defined for all values of n, even when the predicate p is true at only a finite number of points of concrete time.

If, however, the signal p is true infinitely often, then for all n there will exist a time t such that Mapped p n t is true, and this time will be unique. Thus, if Valid p holds, p Proj n will in fact denote the unique time at which p is true for the n^{th} time, as desired. More formally,

$$\vdash \forall p\ n.\ \mathsf{Valid}\ p \supset \mathsf{Mapped}\ p\ n\ (p\ \mathsf{Proj}\ n)$$

which gives the following theorem,

$$\vdash \forall p\ n.\ \mathsf{Valid}\ p\ \supset\ p\,(p\ \mathsf{Proj}\ n)$$

This means that if p is true infinitely often; in other words it holds for all t, then it is also true on its n^{th} projection. That is p Proj n always denotes a point of concrete time at which p is true, and p Proj projects 0 to the first time at which p is true. It follows that the two lemmas about the well-ordering properties of the time sequences can also be proved,

$$\vdash \forall p\ n.\ \mathsf{Valid}\ p\ \supset\ p\ \mathsf{Proj}\ n < p\ \mathsf{Proj}\ (\mathsf{SUC}\ n)$$

$$\vdash \forall p\ n.\ \mathsf{Valid}\ p \supset \forall t.\ p\ \mathsf{Proj}\ n < t \wedge t < p\ \mathsf{Proj}\ (\mathsf{SUC}\ n) \supset \neg p\ t$$

The lemmas can be conveniently collected together using the predicate Last defined previously, to give the following theorem,

$$\begin{aligned}&\vdash \forall p\ r.\ \exists t.\ p\ t\ \wedge\\ &\quad \forall t.\ p\ t \supset \exists m.\ \mathsf{Last}\ p\ (t, t+m) \wedge r(t, t+m) \supset \forall n.\ r(p\ \mathsf{Proj}\ n, p\ \mathsf{Proj}\ \mathsf{SUC}\ n)\end{aligned}$$

This means that for any predicate p and a next-state relation r, this theorem can be used to reduce the problem of establishing,

$$\forall n.\, r\ (p\ \mathsf{Proj}\ n, p\ \mathsf{Proj}\ \mathsf{SUC}\ n)$$

to a pair of simpler problems,

$$\vdash \exists t.\, p\ t$$

$$\vdash \forall t.\, p\ t \supset \exists m.\, \mathsf{Last}\ p\ (t, t+m) \wedge r(t, t+m)$$

A number of related ideas can be defined in terms of projection operators. For instance, the relation p When b is true if p holds in the interval made up of just those states in which the Boolean expression b is true. Thus, the relation

$$(X\ \mathsf{Eq}\ 0)\ \mathsf{When}\ (\mathsf{Len}\ X\ 7)$$

means that the variable X is zero at regular intervals of seven units of time.

It should be noted that Len $(p\ n)\ t$ is true if and only if the predicate p holds at sometime n^{th} in the past.

The definition of p When b is as follows; successive intervals with b true are picked out from the projection of p from the abstract time scale to the concrete time scale,

$$\vdash \forall p\ b.\, p\ \mathsf{When}\ b = \forall n.\, b\ (p\ \mathsf{Proj}\ n)$$

The definition of When is particularly useful in describing the simultaneous actions of parallel processes, such as those in the definition of interrupts and timeouts which are described in the following sections.

5.3.2 Interrupts

Interrupt handling is a familiar problem in real-time programming when an interrupt occurs, perhaps because of a device requiring attention, the running program is suspended and execution begins in the appropriate interrupt service routine. When the service routine finishes, execution of the interrupted program resumes. This is just the kind of behaviour produced by temporal projection. The servicing of the interrupt occurs between two consecutive states of the interrupted program.

Let us write p Upon b to mean that whenever the Boolean expression b is true, the execution of p is interrupted by p'. More formally,

$$\vdash \forall p\ b.\, p \text{ Upon } b = \lambda p'\ t.\, p \text{ When } b \Rightarrow p'\ t \mid p\ t$$

The function p Upon b takes t as its argument, so that it evaluates over the whole interval of time, and returns true if b is true at t.

An interruption is effected by projecting the corresponding formula onto p. For example, the relation

$$pgm \text{ Upon } Printer\ fill\_buf\ (PrintBuf)$$

might specify that whenever the signal $Printer$ is set, the running program is interrupted whilst the $PrintBuf$ is filled.

Interrupts may be nested by simply projecting those of higher-priority onto those of lower-priority. In a specification of the form

$$(prog \text{ Upon } LowPri \ldots) \text{ Upon } HighPri \ldots,$$

the interrupt $HighPri$ has priority over $LowPri$, which in turn has priority over normal processing.

Note that the operator Upon is useful in other situations besides the description of interrupt behaviour. It is used in the specification of the DBW controller, described in Chapter 6, to add timing details to the specification.

5.3.3 Time limits

Sometimes it is necessary to limit the time spent waiting for a particular condition to become true. A familiar real-time programming device which does this is the Timeout. For example, when two processes are exchanging messages, a failure in one process might cause the other to hang waiting for input. In this situation, the working process can recover by giving up after a pre-arranged time limit has expired.

The simplest form of Timeout is expressed by the predicate Timeout (t, b) which waits at most t units of time for the Boolean expression b to become true. It satisfies the inductive property below,

$$\vdash \mathsf{Timeout}\ 0\ b = \mathsf{T}\ \wedge$$
$$\mathsf{Timeout}\ (\mathsf{SUC}\ t)\ b = b \Rightarrow \mathsf{F} \mid \mathsf{Timeout}\ t\ b$$

Another kind of time limit occurs in the specification of real-time systems when one needs to ensure that a condition does become true within a particular interval of time. The construct b Within p takes care of this case. The expression b becomes true within an interval defined by p if there is no such time interval on which p holds but b is not true at some time.

$$\vdash \forall b\ p.\ b\ \mathsf{Within}\ p = \lambda t. \neg(p\ t\ \wedge\ \mathsf{Sometime}\ b\ t)$$

The predicate p defines an interval within which p is sometimes true. For instance, the relation

$$\mathsf{Always}\ (\mathit{request} \longrightarrow \mathit{service}\ \mathsf{Within}\ (\mathsf{Len}\ \mathit{request}\ t'))\ t$$

indicates that a particular *request*, which occurs at time t is always serviced within t' units of time of being registered.

However, it is often the case that a request is only serviced within a fixed time provided that no other requests intervene. This can be formulated by combining the operators Within and When,

$$\vdash \mathsf{Always}\ (\mathit{request} \longrightarrow (\mathit{service}\ \mathsf{WithinLen}\ \mathit{request}\ t')\ \mathsf{When\ Not}\ \mathit{request}')\ t$$

The *request* is serviced within t' time steps on which the alternative *request'* is not registered.

5.4 Formulation of Temporal Correctness

Having formally defined temporal functions, and shown that they construct a well defined time mapping for any predicate p that is true at an infinite number of points of concrete time, it is possible to formulate correctness theorem based on temporal abstraction by time mapping.

If p is an appropriate predicate that indicates which points of concrete time correspond to points of abstract time, then a correctness theorem that relates a design model $M[sig_1, ..., sig_n]$ to an abstract specification $S[s_1, ..., s_n]$ can be formulated as follows,

$$\vdash M[sig_1, ..., sig_n] \supset S[sig_1 \textsf{ When } p, ..., sig_n \textsf{ When } p]$$

This correctness theorem states that whenever the signals $sig_1, ..., sig_n$ satisfy the model, the abstract states $s_1, ..., s_n$ constructed by projecting on $sig_1, ..., sig_n$ when the predicate p is true will satisfy the temporally abstract specification. In general, the predicate p can be defined in terms of the variables $sig_1, ..., sig_n$ to make the times at which the values in the model are projected depend on the behaviour of the device itself. This approach is illustrated in a case study described in Chapter 6. Related work on the verification of temporal correctness using the time mapping approach can be found in [56,69,79].

5.5 Discussion

The literature contains a number of publications on temporal abstraction and verification. The work most closely related to the particular formulation of temporal abstraction by time mapping developed in this chapter is the work on temporal abstraction reported by Herbert [56] and Melham [79]. Herbert defines a higher order predicate UP_OF *ck n t* which is equivalent to the following instance of the more general Mapped predicate defined in Section 5.3.1. Herbert also defines a mapping function ABS by the equation,

$$\vdash \mathsf{ABS}\ select\ sig\ n = sig\ (\varepsilon\lambda.\ select\ n\ t)$$

and uses this function to construct an abstract signal,

$$\mathsf{ABS}(\mathsf{UP\_OF}\ ck)\ sig$$

by sampling the signal *sig* on the rising edges of the clock *ck*. This construction is equivalent to projecting the signal *sig* using the When operator defined in Section 5.3.1 as follows,

$$sig\ \mathsf{When}\ (\mathsf{Rise}\ ck)$$

The main difference between these two formulations is that all the constructs for temporal abstraction by mapping defined in this chapter (e.g. p' When p) are parameterised by a predicate p that identifies the points of time at which a process p' is mapped, while the UP_OF construct which forms the basis of Herbert's work is a specialised predicate for sampling only on the rising edges of a clock.

Melham in [79] also defines a similar function TimeOf and a predicate IsTimeOf which are adopted from the the work on hardware verification using higher order logic discussed by Dhingra [37] and Joyce [69]. Melham uses TimeOf to relate temporal abstraction of the device at different levels rather than on detailed timing analysis. Finally, the two approaches are very similar, but the constructs for temporal abstraction defined in the present work (Mapped, When, etc.) are more general than those defined by Melham in [79]. These temporal constructs are based on the mapping between processes rather than sampling between different time scales. Furthermore, temporal abstraction relationships of the kind involving the time marker function Len and interrupt operators such as Upon and Timeout defined in Section 5.3.2 are not considered in Melham's work. The following chapter illustrates how these temporal constructs can be embedded into a formal specification of a real-time control system.

Chapter 6

A Throttle Control System

6.1 Introduction

The value of formal methods for the specification and verification of real-time systems is well appreciated [95,117]. The purpose of this chapter is to show how its value to the designer of non-trivial control systems is enhanced when mathematical logic is used. A particular example, the controller for a car throttle Drive-By-Wire [109,110,111] is chosen for illustration. The controller is specified in HOL, and from that specification is drawn a prototype which has been implemented in practice. The prototype system has been implemented and tested [1], and could be formally proved to meet its specification. Most essential aspects of the system's behaviour are modelled.

The development of a formal specification for the DBW controller is undertaken in four parts. The first part describes the external interfaces to the controller, giving first an informal description and then a formal representation. The second part gives a formal specification of the controller in terms of its interface. The third part discusses some ways of improving the system and the last part concentrates on verification problems.

[1]The system was developed in conjunction with T&N Technology Ltd. and Econocruise Ltd. in Rugby, England

6.1.1 System description

The controller is to govern the operation of a throttle in a car's carburettor which in turn controls the passage of the fuel mixture. It interacts with the environment through a number of external interfaces,

1. **Accelerator pedal**: the system interacts with the accelerator via a position sensor unit, which is basically a linear potentiometer and an idle switch to calibrate the pedal demand when the system starts up. The idle switch provides a binary signal, namely *idle*, which is true when the pedal demand is zero and false otherwise.

2. **Brake**: the brake input is also a binary switch, designated *brake*, which is true when the brake is applied and false otherwise.

3. **Gear**: the system senses the gear status by an input signal *gear*, which is set to true when the car is in top gear and false otherwise.

4. **Control buttons**: different functions of the controller can be activated by buttons on the control panel. There are three control buttons. All controlling functions are activated by pressing the appropriate buttons, and this will consequently set the control flags to true if certain pre-defined constraints are satisfied. For example, the button to request cruising is denoted by the Boolean signal *cruise*. The cruise control function is to take over the task of maintaining a constant speed when commanded to do so by the driver. When the button is pressed, *cruise* is set to true only when the engine is running and the transmission is in top gear. When the cruise control is activated, the system selects the current speeds, and holds the car at that speed. Deactivate cruising returns control to the driver regardless of any other commands.

   On the control panel, there is also a resume button which causes the system to return the car to the speed selected prior to braking or gear shifting. Function *resume* is also denoted as a Boolean signal.

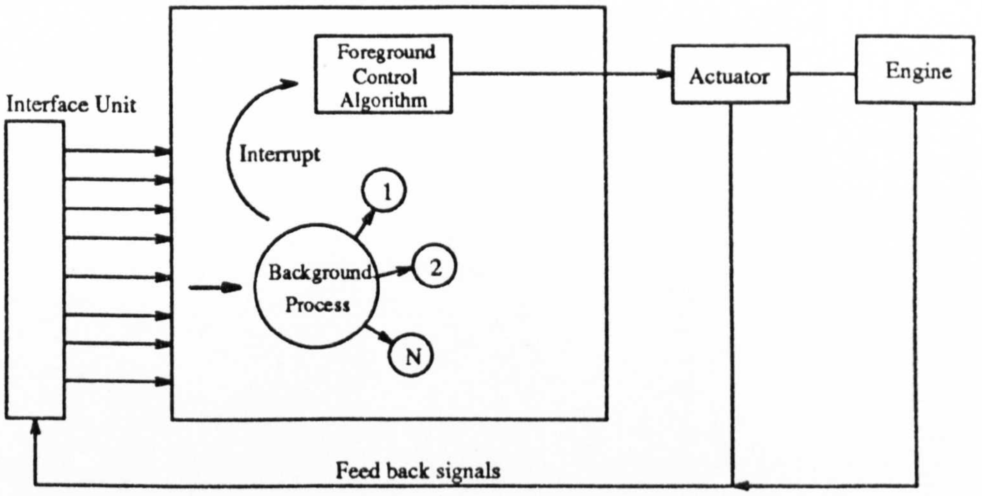


Figure 6.1: DBW system overview

The *traction* control is a Boolean signal which activates the traction mode of the controller when it is pressed and the car is stationary.

5. **Actuator :** the system controls the car through an actuator attached to a throttle. This actuator is mechanically in parallel with the accelerator pedal mechanism, such that whichever one is demanding greater speed controls the throttle. The system is to drive the actuator by means of an electrical signal having a linear relationship with throttle deflection, with 0 volts setting the throttle closed and 16 volts setting it fully open.

6. **Speed sensor unit :** the controller is to measure speed by counting the pulses it receives from a sensor on the drive shaft. The number of pulses per second corresponds to vehicle miles per hour through a proportionality constant.

6.1.2 System requirements

The system requirements include a number of safety and timing constraints. These constraints may be specified as follows,

Safety

- The most important safety behaviour of the controller is that the system should never fail with its actuator being stuck opened. The safety property requires the proof that the system is safe in all modes of operation. More formally, to prove the safety property of the system, an inductive mechanism in HOL of the form,

$$\begin{aligned}\vdash \forall i.\ & \mathsf{SAFE}_0 \land \\ & \mathsf{SAFE}_i \supset \mathsf{SAFE}_{i+1} \land \\ & \mathsf{FAIL}_i \supset \mathsf{SAFE}_i\end{aligned}$$

  is required [2]. The first part of the theorem asserts that the system always starts from an initial state 0 which is safe and from its state machine definition it should show inductively that if its i^{th} state is safe then the next $(i+1)^{th}$ state should also be safe. The last part of the safety theorem ensures that the system should never fail with its actuator fully opened.

Timing characteristics

- When the system senses that the speed is above the selected speed, it must completely release the throttle (this situation would occur when driving downhill). At any speed below this, it must drive the throttle to a deflection proportional to the speed error until it is below the selected speed, the throttle is fully open (the steep uphill situation). The system thus serves as the feedback or control part of a servo loop, in which the engine is the feed-forward part. For smooth and stable servo operation, the system must update its outputs at least once every second.

- To avoid rapid increases in acceleration, the actuator must never traverse its full range in less than 10 seconds. This means the system has a maximum

[2]The subscript indicates the state i^{th} of the system.

response time of T_{res} to respond to a pedal demand. It may close at any rate, however, since the car just coasts when the throttle is closed. The automotive engineers have determined that, with these characteristics, the system will hold the car within $\pm$ 1 mph of the selected speed on a normal gradient, and will give a smooth, and comfortable ride.

- When the car is accelerating, the control system must measure the acceleration and clamp it at 1 mph/sec. Again, the throttle setting will be affected by the gradient. If the acceleration reaches 1.2 mph/sec, the throttle should be closed; at 0.8 mph/sec, it should be fully open. Between these limits, the opening is to be linearly related to acceleration. More formally, the actuator characteristics should be defined such that it should not travel between two consecutive angular position faster than a pre-defined transit time T_{tr}.

- The movement of the actuator should be modelled in such a way that its motions are safe. For example, the throttle should not move forward if there are no requests to go faster, and neither should it close down if there are no requests to do so.

6.1.3 Design methodology

Fig.6.1. shows the overview of the DBW and its interfaces. The rest of the section formalises this view. The design methodology follows these steps,

1. Model the control plant using finite-state machine theory. This is not a formal step in the sense that it relies on the intuition of the designer to represent the *real* world by a formal representation.

2. Write HOL formulae specifying the required control behaviour. The specification may refer to any of the events, data variables and activities of the plant.

3. The specification is then verified to ensure that it satisfies the required safety and timing constraints.

It is clear from the system description that the controller requires a means to output a pulse-width modulation (pwm) signal to the actuator at a regular interval such that it would give a smooth and comfortable ride. This can most easily be achieved by using the temporal interrupt Upon operator described in section 5.3.2. Therefore, the control behaviour of DBW is considered in two different aspects, the background behaviour and the foreground behaviour. The background control task is modelled as a finite-state machine where decision making is essential. Whilst the foreground process, which includes timer interrupts, affects the dynamic behaviour of the system. The operations of the background and foreground processes can best be explained from a programming point of view, where the device cycles indefinitely in a background loops while the foreground process interferes the operation at pre-defined intervals. Thus, the behavioural specification for the DBW controller can generally be defined by a relation of the form,

$$\vdash \mathsf{DBW\_BEHAV} = \mathsf{BACK\_BEHAV}\ \mathsf{Upon}\ (timer)\ \mathsf{FORE\_BEHAV}$$

This means that the background process specified by the relation BACK_BEHAV is regularly interrupted by a foreground process specified by FORE_BEHAV using a pre-defined interrupt *timer*.

The specification of the DBW is defined by predicates which express relations on time-dependent signals. These predicates are represented, in part, by variables representing physical input and output signals. They may also be parameterised by other signals representing the internal state or external conditions governing the behaviour of the device. Time-dependent signals are modelled as functions from discrete time to signal values. As shown in the following type abbreviation, discrete time is represented by the natural numbers.

new_type_abbrev (‘*time*‘,"*num*")

The top level specification for the DBW controller is based on the transformation of a state vector :

$$(s, state) = (s, mode, valid\_cruise, pwm\_act)$$

where s implies the current status of the input signals. The type of s is represented as an n-tuple where each element corresponds to one of the primitive operations on the signal types.

```
let sig_ty = " : time → bool #   % engine running %
                 time → bool #   % top gear signal %
                 time → bool #   % brake input %
                 time → bool #   % watchdog signal %
                 time → bool #   % winding opencircuit signal %
                 time → bool #   % over temperature signal %
                 time → bool #   % idle signal %
                 time → bool #   % traction button %
                 time → bool #   % cruise button %
                 time → bool #   % resume button %
                 time → num  #   % pedal demand %
                 time → num  #   % road speed %
                 time → num"     % position feedback %
```

Selectors for primitive operations in the type *sig_ty* are defined in the formal theory by composing various sequences of the two selectors FST and SND to form projector functions, for instance, the terms GEAR, BRAKE denote projector functions for inputting *gear* and *brake* signal respectively,

$\vdash \forall s.\,\mathsf{GEAR}\ (s : \text{`}sig\_ty) = \mathsf{FST}\ (\mathsf{SND}\ s)$

$\vdash \forall s.\,\mathsf{BRAKE}\ (s : \text{`}sig\_ty) = \mathsf{FST}\ (\mathsf{SND}\ (\mathsf{SND}\ s))$

where the FST and SND are defined as follows,

$\vdash \forall x\ y.\ \mathsf{FST}\ (x, y) = x$

$\vdash \forall x\ y.\ \mathsf{SND}\ (x, y) = y$

The ML variable *sig_ty* is used throughout the formal specification to denote the compound type of the representation variable. When it appears inside a HOL term preceded by a caret ^ (ML anti-quotation), it is expanded to the type expression shown above. This use of ML anti-quotation is unfortunate, but proper type abbreviation cannot be introduced here because this type expression contains polymorphic variables.

The particular ordering of elements in this n-tuple is completely arbitrary. While a less concrete representation could be used, any extra effort in this regard is probably not worthwhile. Details of this particular representation are isolated to a very small part of the formalism and the selector functions for each of the primitive operations are included in Annex C.

The set of thirteen individual control signals, represented by the type *sig_ty*, run from the interface unit to the controller. It is, however, much more convenient to view these signals as a single input to the controller, and once it is inside the system, this bundle of signals is separated into the thirteen individual control signals.

In the formal specification, this bundle can be represented as a signal whose value at any discrete point in time is an n-tuple with thirteen elements. The following definition of the DecodeSig specifies a block of wiring which separates this bundle of control signals into thirteen individual signals by assigning elements of n-tuple representation to corresponding control lines.

$\vdash \mathsf{DecodeSig}\ (sig, runningt, geart, braket, watchoutt, wfailedt, overtempt,$

$idlet, tractiont, cruiset, resumet, demandt, speedt, anglet) =$

$\forall t.\ runningt\ t, geart\ t, braket\ t, watchoutt\ t, wfailedt\ t, overtempt\ t,$

$idlet\ t, tractiont\ t, cruiset\ t, resumet\ t, demandt\ t, speedt\ t, anglet\ t\ = sig\ t$

This definition is expressed in terms of an equation where the left and right hand sides of the equations are n-tuples. Two tuples are equal if and only if they have exactly the same number of elements and matching elements of each n-tuple are both equal in type and in value. In effect, we are using properties of n-tuples to model bit manipulation operations, in particular, the extraction of individual bits from a group of bits.

The *mode* variable defines the system operating modes corresponding to the transition diagram shown in Fig.6.2. There are six discrete states which the FSM can reach. *Mode* is defined as an enumerated variable of the type,

$$
\begin{aligned}
&\vdash \forall e_0\ e_1\ e_2\ e_3\ e_4\ e_5.\ \exists\forall fn.\\
&\quad (fn\ \mathsf{Reset} = e_0)\ \wedge\ (fn\ \mathsf{Idle} = e_1)\ \wedge\\
&\quad (fn\ \mathsf{Traction} = e_2)\ \wedge\ (fn\ \mathsf{Shutdown} = e_3)\ \wedge\\
&\quad (fn\ \mathsf{Manual} = e_4)\ \wedge\ (fn\ \mathsf{Cruise} = e_5)
\end{aligned}
$$

Other state variables, *valid_cruise*, and *pwm_act* are the internal variables which keep tracks of the cruise status flag and the directional directives of the actuator respectively. The directions of the actuator are specified as an enumerated variable of the type *Dir*,

$$
\begin{aligned}
&\vdash \forall e_0\ e_1\ e_2\ e_3.\ \exists\forall fn.\\
&\quad (fn\ \mathsf{Close} = e_0)\ \wedge\ (fn\ \mathsf{Forward} = e_1)\ \wedge\\
&\quad (fn\ \mathsf{Backward} = e_2)\ \wedge\ (fn\ \mathsf{Neutral} = e_3)
\end{aligned}
$$

This means that the movement of the actuator is represented by one of these values, and no other.

6.2 Background Behaviour

Basically, the DBW controller comprises of five functions, namely manual, traction, cruise, shutdown, and idle control. The background behavioural specifica-

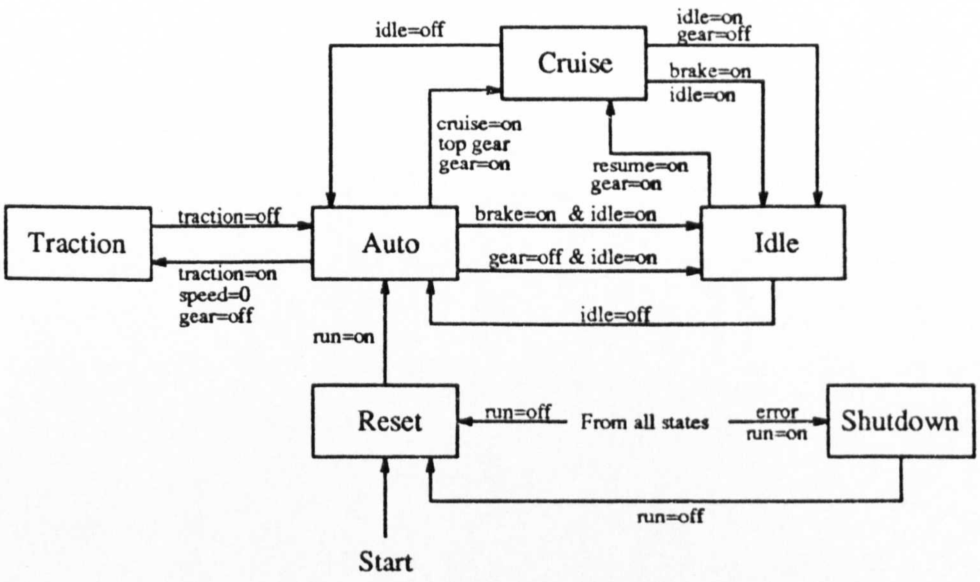


Figure 6.2: DBW State diagram

tion is modelled as a finite-state machine which reads the current status of the input vector s and then delivers its next state. The decision making algorithm is based on a set of pre-defined conditions which correspond to different system operating modes. There are six distinct operating modes which are described as follows,

6.2.1 Manual control

The manual control is the normal working mode of the controller where the position demand which is interpreted from the pedal sensor, is directly output to the actuator. The required input validity conditions are specified by the relation,

$$\vdash \forall b\ p.\ \mathsf{ILLEGAL}\ b\ p = \lambda t.\ \neg\mathsf{Xor2}\ (p, b)\ t$$

The function ILLEGAL is a temporal construct which takes predicates b and p of the type $num \rightarrow bool$ and returns true if b and p occurs simultaneously. For instance,

$$\vdash \mathsf{stop\_ok} = (\mathsf{ILLEGAL}\ idle\ \mathsf{Not}(demand\ \mathsf{Eq}\ 0))\ t$$

The predicate stop_ok defines a validity condition which states that the *idle* switch is on iff [3], at the same time, no *demand* is requested from the pedal sensor.

6.2.2 Cruise/Resume control

The cruise control function is to take over the task of maintaining a constant speed when commanded to do so by the driver. The cruise control can be operated any time the engine is running and the transmission is in top gear. When the driver presses the cruise button to activate cruising, the system selects the current speed, and holds the car at that speed. The cruise control is activated when the input vector satisfies the relation,

$$\begin{aligned}&\vdash \mathsf{C\_COND1}\ (cruise, speed, gear, mode, demand, brake) = \\ &\quad \lambda t.\ (cruise\ t\ \wedge\ \neg(speed\ \mathsf{Eq}\ 0)\ t\ \wedge\ (mode = \mathsf{Manual})\ \vee \\ &\quad \neg(speed\,t\ <\ demand\ t)\ \wedge\ \neg brake\ t\ \wedge\ (mode = \mathsf{Cruise}))\ \wedge\ gear\ t\end{aligned}$$

It is clear from the definition that the driver should be able to increase the speed at any time by depressing the accelerator pedal, or reduce the cruising speed by depressing the brake pedal. Thus, the driver can go faster than the cruise control setting simply by depressing the accelerator pedal far enough. When the pedal is released, the system will regain control. At anytime when the brake pedal is depressed or the transmission shifts out of top gear, the cruise function must be inactive. Following this, when the brake is released, the transmission is back in top gear, and resume is pressed, the system returns the car to the previously selected speed. However, if the cruise function is deactivated during the intervening interval, resume has no effect. Therefore, the condition to activate the cruise control function, by pressing resume, requires that the system is previously in cruise mode and that the cruise control has not been deactivated. Thus, an internal flag is needed to keep track of the cruise status and it can only be modified if cruise is valid. The resume action can then be captured by the predicate valid_c

[3]iff stands for if and only if

as follows,

$$\vdash \mathsf{valid\_c} = (mode = \mathsf{Cruise}) \Rightarrow \neg reset\_ok \mid valid\_cruise$$

Deactivate cruise returns control to the driver regardless of any other commands. However, the cruise condition can be re-activated by the resume control. The resume button causes the system to return the car to the cruising speed selected prior to braking or gear shifting. Therefore, the resumption of cruising can be specified as follows,

$$\vdash \mathsf{C\_COND2}\ (resume, valid\_cruise, gear, mode) =$$
$$\lambda t.\ (resume\ t\ \wedge\ valid\_cruise\ \wedge\ (mode = \mathsf{Idle}))\ \wedge\ gear\ t$$

The two validity conditions for cruising mode can be conveniently combined to give,

$$\vdash \mathsf{C\_COND}\ (cruise, speed, resume, valid\_cruise, gear, mode, demand, brake) =$$
$$\lambda t.\ ((cruise\ t\ \wedge\ \neg(speed\ \mathsf{Eq}\ 0)\ t\ \wedge\ (mode = \mathsf{Manual}))\ \vee$$
$$(\neg(speed\ t\ <\ demand\ t)\ \wedge\ \neg brake\ t\ \wedge\ (mode = \mathsf{Cruise}))\ \vee$$
$$(resume\ t\ \wedge\ valid\_cruise\ \wedge\ (mode = \mathsf{Idle})))\ \wedge\ gear\ t$$

6.2.3 Traction control

The traction mode is activated only when the engine power is taken off for traction or for some other purpose such as running an external pump or air conditioning device. The DBW controller then monitors the engine speed at pre-defined values which may be set by the driver or the manufacturer. The main criteria for this mode to be valid that is the vehicle has to be stationary,

$$\vdash \mathsf{T\_COND}\ (mode, traction, speed, gear, idle) =$$
$$\lambda t.\ ((mode = \mathsf{Manual})\ \vee\ (mode = \mathsf{Traction}))\ \wedge\ traction\ t\ \wedge$$
$$(speed\ \mathsf{Eq}\ 0)\ t\ \wedge\ \neg gear\ t\ \wedge\ \neg idle\ t$$

6.2.4 Idle control

The vehicle is in idle mode if there is no pedal demand or if the brake is not applied. The throttle is then returned to its idle position, i.e. the engine speed is a minimum. In this case the demand is a pre-defined constant which is set by the manufacturer. For the purpose of formal proofs, the actual value of the idle constant is immaterial.

$$\vdash \text{I\_COND}\ (mode, idle, brake, gear) =$$
$$\lambda t.\ ((mode = \text{Manual}) \lor (mode = \text{Cruise}) \lor (mode = \text{Idle})) \land$$
$$idle\ t \land (brake\ t \lor \neg gear\ t)$$

6.2.5 Shutdown/Reset mode

The system is in the shutdown state when either the external watchdog timer has timed out or the engine is overheated or the input state vector is invalid, i.e. the *stop* flag is set,

$$\vdash \text{S\_COND}\ (watchout, overtemp, stop) =$$
$$\lambda t.\ watchout\ t \lor overtemp\ t \lor stop$$

The condition in which the system is reset to its initial state is specified by the relation,

$$\vdash \text{R\_COND}\ running = \lambda t.\ \text{Not}\ running\ t$$

6.2.6 Function NextState

The required combination of these state validity conditions can be conveniently achieved in a function NextState. The function NextState () gives all possible transitions for the controller of the form :

$$\text{NextState}\ (s, state) \rightarrow (state)$$

$\vdash \mathsf{NextState}\ (s, mode, valid\_cruise, pwm\_act) = \lambda t.$

(**let** $running = \mathsf{RUNNING}\ s$ **in**

(**let** $gear = \mathsf{GEAR}\ s$ **in**

(**let** $brake = \mathsf{BRAKE}\ s$ **in**

(**let** $watchout = \mathsf{WATCHOUT}\ s$ **in**

(**let** $wfailed = \mathsf{WFAILED}\ s$ **in**

(**let** $overtemp = \mathsf{OVERTEMP}\ s$ **in**

(**let** $idle = \mathsf{IDLE}\ s$ **in**

(**let** $traction = \mathsf{TRACTION}\ s$ **in**

(**let** $cruise = \mathsf{CRUISE}\ s$ **in**

(**let** $resume = \mathsf{RESUME}\ s$ **in**

(**let** $demand = \mathsf{DEMAND}\ s$ **in**

(**let** $speed = \mathsf{SPEED}\ s$ **in**

(**let** $angle = \mathsf{ANGLE}\ s$ **in**

(**let** $stop\_ok = (\mathsf{ILLEGAL}\ idle\ (\mathsf{Not}\ (demand\ \mathsf{Eq}\ 0)))\ t$

$\lor\ (wfailedt)\ \lor\ (mode = \mathsf{Shutdown})$ **in**

(**let** $reset\_ok = \mathsf{R\_COND}\ (running)\ t$ **in**

(**let** $cruise\_ok = \neg\ reset\_ok\ \land$

$\mathsf{C\_COND}\ (cruise, speed, resume, valid\_cruise, gear, mode, demand, brake) t$ **in**

(**let** $traction\_ok = \neg\ reset\_ok\ \land\ \mathsf{T\_COND}\ (mode, traction, speed, gear, idle)\ t$ **in**

(**let** $shutdown\_ok = \mathsf{S\_COND}\ (watchout, overtemp, stop\_ok)\ t$ **in**

(**let** $idle\_ok = \neg reset\_ok\ \land\ \mathsf{I\_COND}\ (mode, idle, brake, gear)\ t$ **in**

(**let** $valid\_c = ((mode = \mathsf{Cruise})\ \Rightarrow\ \neg reset\_ok\ \mid\ valid\_cruise)$ **in**

$reset\_ok\ \Rightarrow\ (\mathsf{Reset}, valid\_c, \mathsf{PWM}\ (pwm\_act, angle\ t, zerodem))\ \mid$

$shutdown\_ok\ \Rightarrow\ (\mathsf{Shutdown}, valid\_c,\ \mathsf{P\ WM}\ (pwm\_act, angle\ t, zerodem))\ \mid$

$idle\_ok\ \Rightarrow\ (\mathsf{Idle}, valid\_c, \mathsf{PWM}\ (pwm\_act, angle\ t, idledem))\ \mid$

$cruise\_ok\ \Rightarrow\ (\mathsf{Cruise}, valid\_c, \mathsf{PWM}\ (pwm\_act, angle\ t, speed\ t))\ \mid$

$traction\_ok\ \Rightarrow\ (\mathsf{Traction}, valid\_c, \mathsf{PWM}\ (pwm\_act, angle\ t, demand\ t))\ \mid$

$(\mathsf{Manual}, \mathsf{F}, \mathsf{PWM}\ (pwm\_act, angle\ t, demand\ t))))))))))))))))))))))$

The function NextState specifies the overall control mechanism for determining what happens at any instant of time. In the definition of NextState input vectors are used to select the ne:t state and determine the current output of the machine, hence the function NextState is specified in such a way that the state vector only includes all internal variables. Thus a sequence of applications of the function NextState can be represented as a Finite State Machine, where values of s_i represents the input vector at time i,

$$\ldots\mathsf{NextState}\ (s_2, \mathsf{NextState}\ (s_1, \mathsf{NextState}\ (s_0, state_0)))$$

It should be noted that the function NextState delivers a new machine state which depends upon the status of the current input vector and internal flags *mode*, *valid_cruise* and *pwm_act*. The *pwm_act* flag indicates the directional directive of the actuator, and it is defined by the relation PWM. However, a smooth transition of the actuator movement requires that its direction of travel should be changed in such a way that it can not be moving forwards one moment and backwards the next, and vice versa, without ever being neutral, i.e. stand still, for a while. The function PWM can thus be defined as,

$$\begin{array}{l}
\vdash \mathsf{PWM}\ (pwm\_act, ang, dem) = \\
\quad (dem = 0) \Rightarrow \mathsf{Close}\ \mid \\
\quad (dem > ang) \Rightarrow (pwm\_act = \mathsf{Forward}) \Rightarrow \mathsf{Neutral}\ \mid\ \mathsf{Forward}\ \mid \\
\quad (ang > dem) \Rightarrow (pwm\_act = \mathsf{Backward}) \Rightarrow \mathsf{Neutral}\ \mid\ \mathsf{Backward}\ \mid \\
\quad \mathsf{Neutral}
\end{array}$$

This means that if the previous actuator direction of travel is Forward (or Backward) then the throttle needs to stop before it can change to the opposite direction, Backward (or Forward).

Finally, we use the function NextState to define a predicate BACK_BEHAV which specifies the intended behaviour of the controller relative to time-dependent

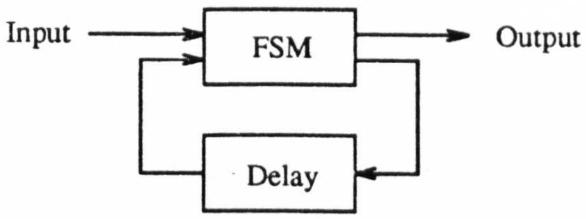


Figure 6.3: A Mealy machine

signals *sigt, modet, valid_ct,* and *pwm_actt.*

$$\vdash \mathsf{BACK\_BEHAV}\ (sigt, modet, valid\_ct, pwm\_actt) =$$

$$\lambda t.\ \mathsf{NextState}\ (sigt\ t, modet\ t, valid\_ct\ t, pwm\_actt\ t)\ t =$$

$$(modet\ (t+1), valid\_ct\ (t+1), pwm\_actt\ (t+1))$$

It is clear from the definition of NextState that the interpretation algorithm implemented by the control unit is based on conditional instructions, that is, a Mealy machine approach in contrast to a Moore machine approach which is based on conditional branches, as shown in Fig.6.3 [4].

Possible transitions of the state machine are shown in Fig.6.4. The transition diagram indicates the possible next state corresponding to a particular set of input vectors. These vectors are grouped into sets of inputs called validity conditions. The proof of all fourteen sets of input validity conditions are outlined in Section 6.6.3.

6.3 Foreground Behaviour

The dynamic characteristics of the system is defined such that the system should respond to the driver demand without taking too long to do so. This section is concerned with trying to constrain the behaviour of the controller so that it does a reasonably good job without being too complex. The first few properties are essential to any reasonable system, and the later properties reflect specific design

[4]In the Mealy model, the output function is related to both the current input and the state, while in the Moore model the output is related only to the state, not to the input.

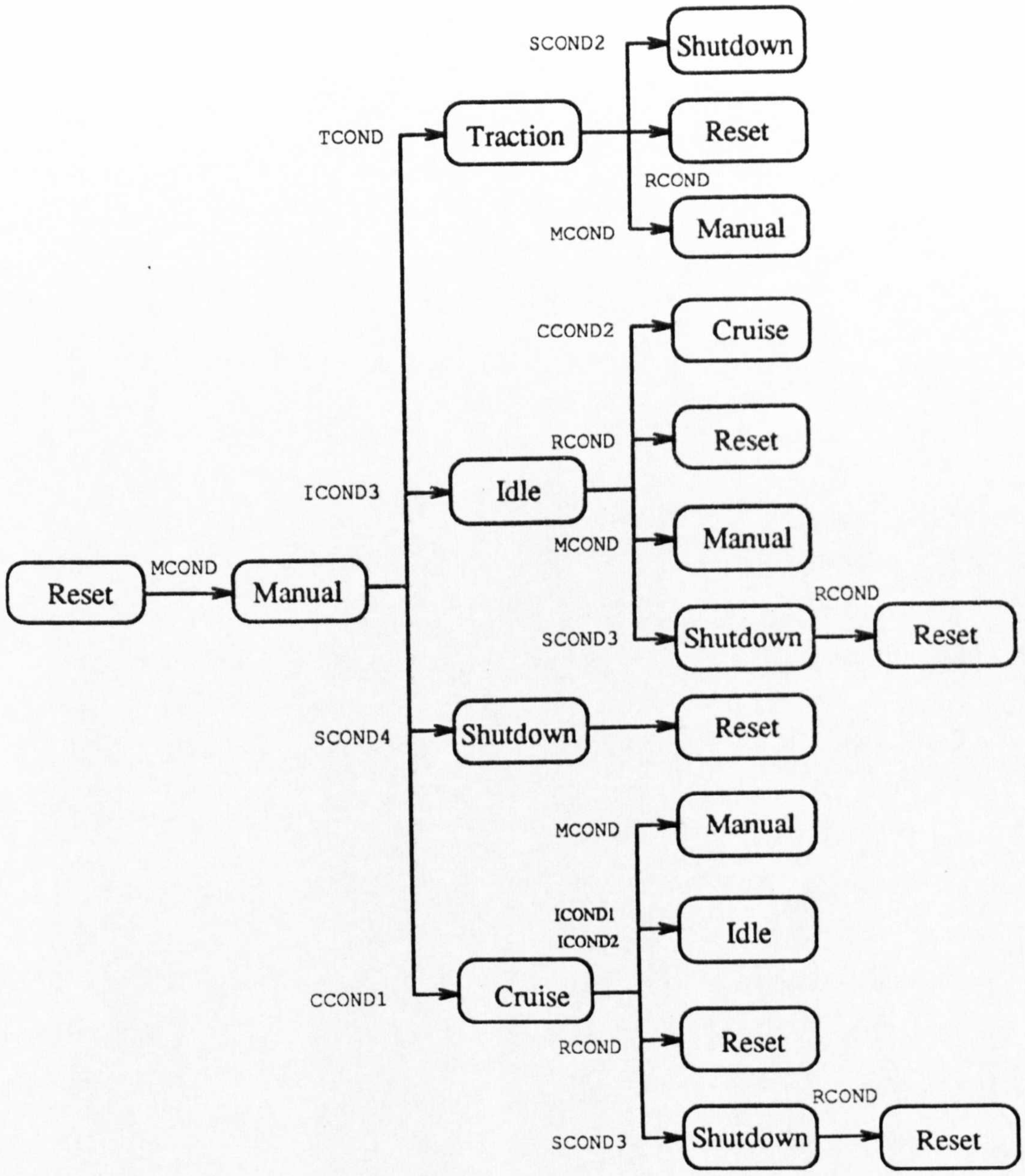


Figure 6.4: DBW state transition diagram

decisions.

6.3.1 Motion

The movement of an actuator's throttle is controlled by two signals, one to switch its motor on and off, the other to control the direction of travel. Predicates act_start and act_stop determine whether the actuator is on or off. They are, of course mutually exclusive,

$$\vdash \forall t.\, \mathsf{Always}\ (\mathsf{Xor2}\ (act\_start, act\_stop))\ t$$

So too are the four temporal directional attributes, *forward, backward, neutral* and *close* which determine whether the throttle is on its way forwards, backwards, stand still, or closing at time *t*,

$$\begin{aligned}\vdash \forall\, t.\ \mathsf{Always}\ (&\mathsf{DecodeCtrl}\ (pwm\_act, forward, backward, neutral,\\ &\qquad close, act\_start, act\_stop)\\ &\supset \mathsf{Xor2}\ (close, \mathsf{Xor3}\ (forward, backward, neutral)))\ t\end{aligned}$$

where the predicate $\mathsf{Xor3}(x, y, z)$ denotes the exclusive disjunction of its three arguments.

$$\vdash \forall x\ y\ z.\, \mathsf{Xor3}\ (x, y, z) = \lambda t.\, (x\ t \lor y\ t \lor z\ t) \land \neg(x\ t \land y\ t \lor y\ t \land z\ t \lor z\ t \land x\ t)$$

This means that exactly one of *x*, *y*, and *z* is true at any instant in time. The exclusive disjunction of two arguments, $\mathsf{Xor2}(x, y)$ is the same as $\mathsf{Xor3}(x, y, \mathsf{False})$.

$$\vdash \forall t.\, \mathsf{Xor2}\ (x, y)\ t = \mathsf{Xor3}\ (x, y, \mathsf{False})\ t$$

It should be noted that these four directive signals are decoded from the *pwm_act* control flag in the background process using the relation,

$$\vdash \mathsf{DecodeCtrl}\ (pwm\_act, forward, backward, neutral, close) =$$

$$(\lambda t.\,(\mathit{forward}\, t = (\mathit{pwm\_act} = \mathsf{Forward})) \wedge$$
$$(\mathit{backward}\, t = (\mathit{pwm\_act} = \mathsf{Backward})) \wedge$$
$$(\mathit{neutral}\ t = (\mathit{pwm\_act} = \mathsf{Neutral})) \wedge$$
$$(\mathit{close}\ t = (\mathit{pwm\_act} = \mathsf{Close})))$$

That is, the actual movement of the actuator at any time t is controlled indirectly by an appropriate set of input state vectors and the internal control flags of the machine via the background process.

Consequently, at any instant in time t, the car is said to be accelerating when its actuator is on and the throttle is moving forwards, and decelerating when its actuator is on and the throttle is moving backwards. It should be noted that the concept of acceleration and deceleration presented here is purely in terms of the control system point of view. They only define the behaviour of the actuator at a particular instant in time. Hence,

$$\vdash \mathsf{Accelerate}\ (\mathit{act\_start}, \mathit{forward}) = \mathit{act\_start}\ \mathsf{And}\ \mathit{forward}$$
$$\vdash \mathsf{Decelerate}\ (\mathit{act\_start}, \mathit{backward}) = \mathit{act\_start}\ \mathsf{And}\ \mathit{backward}$$

It should be noted that the predicate Accelerate and Decelerate are temporal constructs of the type $\mathit{num} \rightarrow \mathit{bool}$.

6.3.2 Real-time characteristics

The most important property of a throttle controller is that it takes the car to a speed that the driver wants it to go; that is, all demands are eventually achieved. For example, if a request for full throttle is registered from the accelerator pedal, then the throttle eventually opens to its full extent. This means that an appropriate action will be executed at sometime instant in time, i.e.

$$\vdash \mathsf{Always}\ (\mathit{request} \longrightarrow \mathsf{Sometime}\ (\mathit{action}))$$

Likewise, a request from the pedal is eventually rewarded with the appropriate service,

$$\begin{aligned}\vdash \forall t.\ & \mathit{forward}\,t \Rightarrow \mathsf{Sometime}\,(\mathit{output})\,t \mid \\ & \mathit{backward}\,t \Rightarrow \mathsf{Sometime}\,(\mathit{output})\,t \mid \\ & \mathit{neutral}\ t\end{aligned}$$

The trouble is that for most practical purposes these properties are much too weak. They allow a throttle to remain idle for a very long time, it could be so long that it could considerably affect the overall performance of the car [5]. Therefore, the following constraints will ensure that the dynamic behaviour of the controller is satisfactory.

6.3.3 Maximum response time

To avoid rapid increases in acceleration, the actuator must never traverse its full range in less than 10 seconds. It may close at any rate, however, since the car just coasts when the throttle is closed. The automotive engineers have determined that with these characteristics, the system will hold the car within ± 1 mph of the selected speed on normal gradients, and will give a smooth, comfortable ride.

However, it would be much more useful to put an upper bound T_{res} on the time taken to respond to each throttle demand. A more realistic specification should take account of this timing requirement, which is most simply done by counting time when the actuator is on,

$$\begin{aligned}\vdash\ & \mathsf{RESPONSE}\ (\mathit{forward}, \mathit{backward}, \mathit{neutral}, \mathit{output}, \mathit{act\_start}) = \\ & \quad \lambda t.\ \mathit{forward}\,t \Rightarrow (\mathit{output}\ \mathsf{Within}\ \mathsf{Len}\ \mathit{act\_start}\ T_{res})\ \mathsf{When}\ \mathit{act\_start} \mid \\ & \quad\quad \mathit{backward}\,t \Rightarrow (\mathit{output}\ \mathsf{Within}\ \mathsf{Len}\ \mathit{act\_start}\ T_{res})\ \mathsf{When}\ \mathit{act\_start} \mid \\ & \quad\quad \mathit{neutral}\,t \end{aligned} \tag{6.1}$$

[5]There is also a technical difficulty with termination because Sometime, as we have it, is a strong operator which asserts that its argument surely does hold at some time.

where the operator Within is defined in section 5.3.3. The time limit T_{res} is the maximum time for which a throttle can be moving before it reaches the demanded position. Naturally the constraint only applies when the system is in service. When it is out of service or shutdown, that is when the *stop* flag is set, nothing at all will affect the behaviour of the controller until the *stop* flag has been reset by an external agent, probably a service engineer in practice.

The problem has now been transformed into one of ensuring that the actuator does not remain stationary forever, ie. it is stuck. This is the purpose of the next constraint.

6.3.4 Maximum registered time

Consider what happens when the car is stationary with its actuator off. What makes it ever start moving ? Hopefully, when there are demands, and the system is not out of service, then it will begin to move after waiting a maximum of T_{reg} time for all other components to be initialised.

$$\begin{aligned}
&\vdash \mathsf{REGISTER}\ (act\_stop, act\_start, neutral) = \\
&\quad \lambda t.\ \mathsf{Always}\ (act\_stop\ \mathsf{And\ Not}\ neutral \\
&\qquad \longrightarrow act\_start\ \mathsf{Within\ Len}\ (\mathsf{Not}\ neutral)\ \ : T_{reg})\ t
\end{aligned} \tag{6.2}$$

6.3.5 Maximum transit time

In order to calculate an upper bound for the maximum response time T_{res}, let us suppose that the controller should take no more than T_{tr} units of time to travel between two consecutive demand positions, that is,

$$\begin{aligned}
&\vdash \mathsf{TRANSIT}\ (accelerating, decelerating) = \\
&\quad \lambda t.\ \forall pos\ ang. \\
&\quad \mathsf{Always}(\\
&\qquad (accelerating\ \mathsf{And\ Not}\ (pos\ \mathsf{Below}\ ang) \\
&\qquad\quad \longrightarrow (\mathsf{Not}\ (pos\ \mathsf{Below}\ ang)\ \mathsf{Within\ Len}\ accelerating\ T_{tr})\ \mathsf{And}
\end{aligned}$$

$$(\textit{decelerating}\ \mathsf{And\ Not}\ (\textit{pos}\ \mathsf{Above}\ \textit{ang}) \\ \longrightarrow (\mathsf{Not}\ (\textit{pos}\ \mathsf{Above}\ \textit{ang})\ \mathsf{Within\ Len}\ \textit{decelerating}\ T_{tr}))\ t \qquad (6.3)$$

The relations Below and Above are temporal infix functions which evaluate to true if the actuator position *pos* is equal to $ang - 1$ and $ang + 1$ respectively [6],

$$\vdash \forall pos\ ang. pos\ \mathsf{Below}\ ang = \lambda t.\ (pos\ \mathsf{Eq}\ (\mathsf{PRE}\ ang))\ t$$

$$\vdash \forall pos\ ang. pos\ \mathsf{Above}\ ang = \lambda t.\ (pos\ \mathsf{Eq}\ (\mathsf{SUC}\ ang))\ t$$

In other words, if the throttle is moving forwards and it is at the initial angular position *ang*, then within T_{tr} units of time it will be at least at the next angular position $ang + 1$. Similarly, when going backwards, it will reverse to the next position in T_{tr} time steps.

Suppose that the specification could be realised by having a throttle continously opening and closing, stopping at every degree, repeatedly making an entire round trip, then in this case, the maximum time spent in motion is

$$T_{res} = 2 \times\ (m - 1)\ T_{tr}\ where\ m = 90^{\circ}$$

This is an upper bound on the maximum response time for any well designed system, the average response time ought to be much less than T_{res}.

6.3.6 Position and direction

The relationship between the position of a throttle and its direction of travel are just the obvious ones. That is if its actuator is off then the throttle should recoil to the close position due to the spring action; if it is going forwards then it should be seen to go forwards; that is, the width of the fuel flow must not decrease, and similarly, the width of the passage must not increase when it is going backwards.

$\vdash \mathsf{POS\_DIR}\ (act\_stop, accelerating, decelerating,$

[6] PRE *ang* and SUC *ang* are alternative ways of expressing $(ang - 1)$ and $(ang + 1)$ respectively.

$$\begin{array}{l}
\qquad close, forward, backward, neutral) = \\
\lambda t.\ act\_stop\ t \Rightarrow \mathsf{Next}\ (close)\ t \mid \\
\quad accelerating\ t \Rightarrow forward\ t \Rightarrow \mathsf{Next}\ (forward)\ t \mid \\
\qquad\qquad backward\ t \Rightarrow \mathsf{Next}\ (\mathsf{Not}\ forward)\ t \mid \\
\qquad\qquad \mathsf{Next}\ (neutral)\ t \mid \\
\quad decelerating\ t \Rightarrow forward\ t \Rightarrow \mathsf{Next}\ (\mathsf{Not}\ forward)\ t \mid \\
\qquad\qquad backward\ t \Rightarrow \mathsf{Next}\ (backward)\ t \mid \\
\qquad\qquad \mathsf{Next}\ (neutral)\ t \mid \\
\quad \mathsf{Next}\ (neutral)\ t
\end{array} \qquad (6.4)$$

6.4 Improving the Performance

Conditions (6.1-6.4) do not constrain the controller to be a particularly good one. They are met, as observed above, by a system in which the throttle repeatedly opens and closes, stopping at every degree. The next few constraints are aimed at eliminating some undesirable behaviour by modelling the actuator behaviour as described in Section 6.1.2.

First, suppose that the throttle is stationary. Constraint (6.2) determines the maximum time for which it can remain stationary if there are outstanding demands. However, it should not begin to move if there are no function requests or throttle demands; or if it is held at an angle and when it does begin to move it must decide whether to go forwards or backwards. The only constraint on this choice is that it should not head off forwards if there are no requests to go faster, and neither should it close down if there are no requests to do so.

$$\begin{array}{l}
\vdash \mathsf{PERFORM1}\ (act\_stop, act\_start, accelerating, decelerating, \\
\qquad forward, backward, neutral, close) =
\end{array}$$

$$\begin{aligned}\lambda t.\ \textsf{Always}\ (act\_stop \longrightarrow \textsf{Next}\ (neutral)\ t \Rightarrow\ & act\_stop\ \mid\\ & act\_start\ \textsf{And}\ (\textsf{Not}\ decelerating)\ t \Rightarrow forward\ \mid\\ & act\_start\ \textsf{And}\ (\textsf{Not}\ accelerating)\ t \Rightarrow backward\ \mid\\ & close)\ t\end{aligned}$$

Note that when its actuator is on, the throttle must be moving in some direction,

$$\vdash \textsf{PERFORM2}\ (act\_start, neutral) = \lambda t.\ \textsf{Always}\ (act\_start \longrightarrow \textsf{Not}\ neutral)\ t$$

Next, consider what happens once the throttle has started to move ? How does its position change, and when should it stop ? Constraint (6.3) and (6.4) ensure that within T_{tr} units of time it arrives at the next position. It must then decide whether or not to stop. If there is still a demand at that point then the throttle must stop, but it should not stop if it is not required to do so.

$$\begin{aligned}&\vdash \textsf{PERFORM3}\ (forward, backward, neutral, act\_stop, act\_start) =\\ &\quad \lambda t.\ \textsf{Always}\ (\\ &\qquad forward\ t \Rightarrow \textsf{Next}\ (\textsf{Not}\ forward)\ t \Rightarrow act\_stop \mid forward\ \mid\\ &\qquad backward\ t \Rightarrow \textsf{Next}\ (\textsf{Not}\ backward)\ t \Rightarrow act\_stop \mid backward\ \mid\\ &\qquad neutral\ t \Rightarrow \textsf{Next}\ (\textsf{Not}\ neutral)\ t \Rightarrow act\_start\ \mid\ act\_stop\ \mid\\ &\qquad act\_stop)\ t\end{aligned}$$

This simple method of deciding when to stop, using a finite step size, only works if, as directed by constraint (6.4), the throttle can not swing past a position without ever being at it. Otherwise it must have been decided in advance when to stop.

Lastly, in the interest of fairness, the throttle should keep going in the same direction for as long as possible. If it is idle then it should remain so until there are demands to respond to.

$$\begin{aligned}&\vdash \textsf{PERFORM4}\ (forward, backward, neutral, accelerating, decelerating) =\\ &\quad \lambda t.\ \textsf{Always}\ (\end{aligned}$$

$$\begin{aligned}&\mathit{forward}\ t \Rightarrow \mathsf{Next}\ (\mathit{accelerating}\ t \Rightarrow \mathit{forward} \mid \mathit{neutral}) \mid \\ &\mathit{backward}\ t \Rightarrow \mathsf{Next}\ (\mathit{decelerating}\ t \Rightarrow \mathit{backward} \mid \mathit{neutral} \mid \\ &\qquad\qquad \mathsf{Next}\ \mathit{neutral})\ t\end{aligned}$$

This decision only comes into effect when the throttle is stopped at a position because PERFORM3 prevents the throttle from ever changing direction unless it is stopped.

The above constraints can be combined to specify the foreground process which is defined by the relation FORE_BEHAV,

$$\begin{aligned}&\vdash \mathsf{FORE\_BEHAV}\ (\mathit{pwm\_act}, \mathit{output}) = \\ &\lambda t.\ \forall \mathit{act\_start}\ \mathit{act\_stop}\ \mathit{close}\ \mathit{forward}\ \mathit{backward}\ \mathit{neutral}. \\ &\quad (\mathbf{let}\ \mathit{accelerating} = \mathsf{Accelerate}\ (\mathit{act\_start}, \mathit{forward})\ \mathbf{in} \\ &\quad (\mathbf{let}\ \mathit{decelerating} = \mathsf{Decelerate}\ (\mathit{act\_start}, \mathit{backward})\ \mathbf{in} \\ &\quad \mathsf{DecodeCtrl}\ (\mathit{pwm\_act}, \mathit{forward}, \mathit{backward}, \mathit{neutral}, \mathit{close})\ \mathsf{And} \\ &\quad \mathsf{RESPONSE}\ (\mathit{forward}, \mathit{backward}, \mathit{neutral}, \mathit{output}, \mathit{act\_start})\ \mathsf{And} \\ &\quad \mathsf{REGISTER}\ (\mathit{act\_stop}, \mathit{act\_start}, \mathit{neutral})\ \mathsf{And} \\ &\quad \mathsf{TRANSIT}\ (\mathit{accelerating}, \mathit{decelerating})\ \mathsf{And} \\ &\quad \mathsf{POS\_DIR}\ (\mathit{act\_stop}, \mathit{accelerating}, \mathit{decelerating}, \mathit{close}, \\ &\qquad\qquad \mathit{forward}, \mathit{backward}, \mathit{neutral})\ \mathsf{And} \\ &\quad \mathsf{PERFORM1}\ (\mathit{act\_stop}, \mathit{act\_start}, \mathit{accelerating}, \mathit{decelerating}, \\ &\qquad\qquad \mathit{forward}, \mathit{backward}, \mathit{neutral}, \mathit{close})\ \mathsf{And} \\ &\quad \mathsf{PERFORM2}\ (\mathit{act\_start}, \mathit{neutral})\ \mathsf{And} \\ &\quad \mathsf{PERFORM3}\ (\mathit{forward}, \mathit{backward}, \mathit{neutral}, \mathit{act\_stop}, \mathit{act\_start})\ \mathsf{And} \\ &\quad \mathsf{PERFORM4}\ (\mathit{forward}, \mathit{backward}, \mathit{neutral}, \mathit{accelerating}, \mathit{decelerating}) \\ &\quad))\ t\end{aligned}$$

The FORE_BEHAV relation is a time-dependent function, therefore at any time *t* it receives a flag *pwm_act* from the background process BACK_BEHAV and delivers an *output* signal to the external actuator whose dynamic characteristics are specified by the timing constraints described above.

6.5 The Complete Specification

The normal behaviour of the controller, when it is in service, is described by a combination of the above constraints. But when the shutdown flag is set those constraints no longer apply and the system goes out of service. However, the total behaviour can be captured in the predicate DBW_BEHAV,

$$\vdash \mathsf{DBW\_BEHAV} = \mathsf{Always}\ (\mathsf{BACK\_BEHAV}\ (sigt, modet, valid_c t, pwm_a ctt)\ \mathsf{Upon}\ timer\ \mathsf{FORE\_BEHAV}\ (pwm_a ctt\ t, output))\ t$$

This means that the controller comprises of two processes [7]. The background process is continually interrupted by the foreground process which controls the actual movement of the actuator.

The additional condition, which is required to make the system functional, is that the *timer* interrupt must be true infinitely often, that is, it satisfies the predicate,

$$\vdash \forall t.\ \mathsf{Valid}\ timer$$

Thus, the complete DBW specification can be defined as follows,

$$\vdash \mathsf{DBW\_BEHAV} = \mathsf{Always}\ (\mathsf{BACK\_BEHAV}\ (sigt, modet, valid_c t, pwm_a ctt)\ \mathsf{Upon}\ timer\ \mathsf{FORE\_BEHAV}\ (pwm_a ctt\ t, output)\ \mathsf{And}\ \mathsf{Valid}\ timer)\ t$$

6.6 Verification Plan and Methodology

This section describes the logical form used to state correctness results for the DBW controller, a plan to achieve these results and some of the basic proof techniques which are used to carry out this plan in the HOL system.

[7]It should be noted that these processes are not in parallel.

6.6.1 Proof plan

Although the terms *correctness*and *verification* may be understood in an informal context to mean different things to different people, these terms have a precise, technical meaning when formal logic is used to verify a design. The formal verification (or proof of correctness) for the DBW system refers to the derivation of a theorem by formal proof in the HOL formulation of higher-order logic. This theorem relates the specification of the intended behaviour, given by the predicate DBW_BEHAV, to some safety and timing constraints.

The bulk of the formal proof of the DBW controller is organised into two main steps:

1. Prove that the design is safe, i.e. it is required to verify that the behaviour of the system is safe at all times and in all possible states.

2. Prove that each state transition correctly interpretes the specification of the background process. That is the correctness of the state diagram shown in Fig.6.2 must be preserved.

The proof is complete in a third (and a relatively short) step by establishing a formally defined theorem which states that the intended behaviour of the controller satisfies a deterministic finite-state machine relation.

3. Prove that DBW specification is a formally well-defined finite-state machine.

6.6.2 Safety properties

The most important safety rule which applies to the controller design is that, at any instant in time t, the system should never fail with its throttle fully opened. Thus, a state is considered to be safe iff it satisfies the predicate,

$$\vdash \forall s\ m\ v\ p.\ \mathsf{State\_Safe}\ (s,m,v,p) = \lambda t.\ \neg\mathsf{WFAILED}\ s\ t\ \vee\ (p = \mathsf{Close})$$

Following this, the system is safe if it delivers a next set of outputs which belong to a safe operating mode, i.e.

$$\vdash \forall s\ m\ v\ p.\ \mathsf{SAFE}\ (s, m, v, p) = \lambda t.\,\mathsf{State\_Safe}\ (s, \mathsf{NextState}\ (s, m, v, p)\ t)\ t$$

Thus, theorems which define the safe behaviour of the system can be proved as follows,

$$\vdash \forall t.\ \mathsf{Always}\ (\mathsf{Not}\ (\mathsf{RUNNING}\ s)\ \longrightarrow\ \mathsf{SAFE}\ (s, m, v, p))\ t \qquad (6.5)$$

$$\vdash \forall t.\ \mathsf{Always}\ ((\mathsf{WFAILED}\ s\ \mathsf{And}\ \mathsf{RUNNING}\ s)\ \longrightarrow\ \mathsf{SAFE}\ (s, m, v, p))\ t \qquad (6.6)$$

It should be noted that these theorems are effectively stating the behaviour of the system in *static* and *dynamic* failure conditions. That is, the first theorem ensures that if the system is not in service then it is apparently safe, and the second asserts that the system results in a safe state if the actuator failed while the engine is running.

It is clear from the specification of NextState and SAFE that if the system receives an input vector which is safe at time t then it delivers an operating condition which is also safe at time $t + 1$. However, it is not obvious how the controller gets into a safe state at the beginning. Therefore, the following constraint defines the initial or startup mode of the system,

$$\vdash \forall s\ m\ v\ p.\ \mathsf{INIT}\ (s, m, v, p) = \lambda t.\ \neg\mathsf{RUNNING}\ s\ t\ \wedge\ (\mathsf{DEMAND}\ s\ \mathsf{Eq}\ 0)\ t$$

The definition asserts that the controller is initialised whenever the engine is not running and there is no demand from the pedal sensor. It can easily be proved that the following condition always holds,

$$\vdash \forall t.\ \mathsf{Always}\ (\mathsf{INIT}\ (s, m, v, p) \longrightarrow \mathsf{SAFE}\ (s, m, v, p))\ t \qquad (6.7)$$

The problem now is to verify the total safety property of the controller. The proof procedure is outlined as follows,

1. First prove that the initial state is safe, then

2. Use induction to prove that if the system is safe at any time t then it should deliver a next state at time $t + 1$ which is also safe.

The safety property is proved to give the following theorem [8],

$$\begin{aligned} &\vdash \forall t.\ \mathsf{INIT}\,(s\,t, m\,t, v\,t, p\,t)\,t \supset \mathsf{SAFE}\,(s\,t, m\,t, v\,t, p\,t)\,t \wedge \\ &\qquad \mathsf{BACK\_BEHAV}\,(s, m, v, p)\,t \wedge \mathsf{SAFE}\,(s\,t, m\,t, v\,t, p\,t)\,t \\ &\qquad\qquad \supset \mathsf{State\_Safe}\,(s\,t, m\,(t+1), v\,(t+1), p\,(t+1))\,t \end{aligned} \tag{6.8}$$

This asserts that if the system starts up in an initially safe state and, the current mode at any time t is safe then it delivers a next state which is safe at time $t + 1$.

In fact, the theorems (6.5-6.8) provide an inductive mechanism for verifying the total safety behaviour of the system. It has been shown in (6.7) that the controller will startup with a safe state, and then by induction, the theorem (6.8) asserts that the system will also be safe as long as it is running, and finally, theorems (6.5-6.6) ensure that the controller will stop in a safe mode when it is out of service.

6.6.3 State transitions

Verifying the correctness of the DBW at the state level is relatively simple because each state transition is implemented by a fixed sequence (or finite set of sequences) of input vectors.

It is relatively straightforward to reason about a fixed sequence (or a finite set of fixed sequences) of inputs. To derive its cumulative effect, we use inference rules of higher-order logic to *symbolically execute* this sequence of inputs. The term *symbolic execution* is used here in a purely descriptive sense for a proof technique which is actually nothing more than repeatedly unfolding various parts of the specification (or consequence of this specification).

[8]Details of the proof are included in Annex C

Forward proof is used to symbolically execute a sequence of inputs starting with assumptions about the initial state and applying inference rules to derive subsequent states in the computation. Similar techniques were also used to partially verify the VIPER microprocessor [31,32].

The analysis of the way machine states follow each other leads to a representation of the state transitions as a tree (or more accurately, a graph). Each state transition is implemented as a sequence of input events which may effect the internal states or flags of the machine. The sequence is determined as the inputs are read, according to the internal state and the current input vector. The possible sequences define a graph of transition states, each state pointing to one or more others, and one designated Reset state, as shown in Fig.6.4.

The first step of the proof procedure is to show that the system always delivers a new state from a current set of internal variables, i.e. the system is functional at all time, thus,

$$\vdash \forall m\ v\ p\ t.\ \exists s.\ \exists\forall m'\ v'\ p'.\ \mathsf{NextState}\ (s, m, v, p)\, t = (m', v', p')$$

The theorem asserts that given a set of current states, there is at least one set of input which delivers a new operating condition which is uniquely defined by the NextState function.

What has to be proved next is that under certain pre-defined conditions, the functional specification agrees with the intended behaviour. The first thing to establish is what that state is; the second is whether NextState gives that state. All connections in Fig.6.4 are labelled with the corresponding state transition lemmas [9].

Wherever possible, two or more transitions having the same transforming effects are grouped together to form a single transition. For example, the transition R_COND represents a set of transitions from all states having *running* signal

[9]These proofs are included in Annex C

evaluates to False at time t,

$$\begin{aligned}&\vdash \forall t.\ (s = \text{False}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb})\\ &\quad \supset (m = \text{Cruise}) \Rightarrow \text{NextState}\ (s, m, v, p)\ t = (\text{Reset}, \text{F}, \text{Close})\ \mid\\ &\qquad\qquad \text{NextState}\ (s, m, v, p)\ t = (\text{Reset}, v, \text{Close})\end{aligned}$$

The theorem asserts that provided the state vector s satisfies the indicated value then the controller will be reset, internal flags being modified appropriately. Each transition in the graph is therefore characterised by saying how NextState changes a state and selects the next node during that transition [10].

All the state transitions shown above have been proved. The proofs are tedious and messy since there are many cases to consider, however, they follow the same pattern of tactics. Therefore proof procedures are not repeated here but included in Annex C for convenience.

6.6.4 Deterministic state-machine

A state machine is *deterministic* if the next state is uniquely determined by the current state and its input. If all states of the machine are labelled then it is called a labelled-state automata [75].

$$\begin{aligned}&\vdash \forall Q\ N\ e.\ \text{LSA}\ (Q, N)\ e =\\ &\quad \exists s.\ Q\ (e\ 0, s\ 0)\ \land\ \forall t.\ N\ (e\ t, s\ t)\ (e\ (\text{SUC}\ t), s\ (\text{SUC}\ t))\end{aligned}$$

The definition states that a function N is a Labelled-State Automata (LSA) iff there exists an initial condition Q which is true at time 0 and N holds for successive time interval of t. In other words, LSA divides the operating domain of the state machine into two subsets; one set comprises of initial conditions and the other is a set of normal operations which recursively holds for all time t.

Following this, a deterministic machine is safe if it can consequently be clas-

[10]Note that by using the Arb function the state vector s at time t represents a *set* of inputs.

sified into three domains; initial, normal and safety subsets [11].

$$\vdash \forall Q\ P\ N\ e.\ \mathsf{PLSA}\ (Q,P,N)\ e =$$
$$\exists s.\ Q\ (e\ 0, s\ 0)\ \wedge\ \forall t.\ P\ (e\ t, s\ t))\ \wedge\ \forall t.\ N\ (e\ t, s\ t)\ (e\ (\mathsf{SUC}\ t), s\ (\mathsf{SUC}\ t))$$

The equivalence proofs of these assertions are shown in [75] and not repeated here. The main aim is to prove that the state machine behaviour of the DBW satisfies these characteristics. The following theorem has been proved,

$$\vdash \forall ip\ op.\ \mathsf{Init}\ (ip, op)\ \supset\ \mathsf{Safe}\ (ip, op)\ \wedge$$
$$\forall ip\ op\ ip'\ op'.\ \mathsf{Nextstate}\ (ip, op)\ (ip', op')\ \wedge\ \mathsf{Safe}\ (ip, op) \supset \mathsf{Safe}\ (ip', op')$$
$$\supset\ \forall e.\ \mathsf{LSA}\ (\mathsf{Init}, \mathsf{Nextstate})\ e = \mathsf{PLSA}\ (\mathsf{Init}, \mathsf{Safe}, \mathsf{Nextstate})\ e$$

The theorem asserts that provided the initial and all its subsequent states are safe then it is deterministic. In fact, this theorem holds for all states since the safety property of the DBW has been proved in section 6.6.2. The derivation of this proof is shown in Annex C.

6.7 Discussion

In this case study, the FSM/TL [12] framework is used for modelling, specifying and verifying systems composed of real-time discrete event processes. Finite state machines are used to represent event processes, and temporal logic provides the assertions for specification and verification.

The connection between FSM and TL is made via a transition system which consists of a (possibly infinite) state space, a set of transitions defining state transformation, a set of initial states, and a safety family. A computation is expressed as a sequence of states whose initial state is in the initial state set, whose successor states are obtained by applying permitted transitions, and where the safety family

[11] This is called a Partitioned Label State Automata (PLSA).

[12] Finite-State Machine/Temporal Logic

are used to ensure that certain sets of transitions that are continuously enabled or infinitely often enabled in the computation are achieved. The approach used in this case study can form the basis for subsequent behavioural analysis of real-time discrete event systems and its characteristics can be summarised as follows,

- State transitions have associated lower and upper time bounds (measured with respect to a global clock) so that real-time properties (including delays and timeouts) can be represented.
- Two different variables are introduced: a time variable t (the current clock time) and an implicit next-transition variable n (for referring to the events of complex systems).
- Instead of computations, the state transition generates *trajectories* or sequence of states that take into account the lower and upper time bounds of transitions.
- The trajectories can be interpreted in temporal logic which uses a mixture of standard operators together with the time variable to express quantitative timing properties.

The expressive power of higher-order logic has been an advantage in several ways for the work described in this case study. One of the most important uses of higher-order functions and relations has been to parameterise definitions by a representational variable providing a formal basis for our approach to abstract specification.

We have found several instances where the ability to use a relation in place of a function was very desirable. Our model of control signals with relations as an alternative to using combining functions.

A number of errors in the design and specification of DBW were discovered in the course of verifying its correctness. A few of these errors were not simple mistakes and not the kind of error easily revealed by simulation (they were concerned with subtle aspects of the interrupt specification). However, many simple mistakes were also found by verification but they could have been discovered earlier without incurring the high cost of verification if the specifications had been executable. Camilleri [25] points out the advantage of executing formal specifications as a means of supporting a verification methodology. In this respect, the Boyer-Moore approach [14] offers the advantage of directly executable specifications. The increased expressiveness of higher-order logic results in terms which do not always have an obvious executable specification. However, the implementation of an executable subset of higher-order logic is a foreseeable addition to the HOL system [13].

It was extremely useful (in fact, much more useful than we initially expected) to embed some operators from temporal logic in our higher-order framework. It is likely that work in embedding other calculi in higher-order logic will be a particular interesting area of future development in HOL. In addition to Hale's work on Interval Temporal Logic [52], some other examples include work by Gordon on programming logic [51], by Loewenstein on a theory for deterministic and non-deterministic state machines [75], and related work by Joyce on asynchronous memory [69].

[13]Camilleri [25] describes simulation tools for an executable subset of higher-order logic oriented towards hardware simulation.

Chapter 7

Mechanical Solid Modelling

7.1 Introduction

This chapter deals with the specification issues encountered when designing mechanical components. Most mechanical products are collections of rigid solids which are manufactured and assembled via processes whose effects are primarily geometrical. Geometrical information - for specifying parts and products, and for controlling processes - obviously is of crucial importance in the mechanical industries, and techniques for modelling and handling such information are required for automatic or computer-aided design and production. Although there are a number of geometric modelling systems commercially available [9,16,19], most of them rely on the user to verify the completeness, validity, and uniqueness of a model. Such manual verification becomes increasingly more difficult as the model becomes more complex, even with sophisticated computer graphic display techniques. The work presented in this chapter describes an attempt to automate the verification of such modelling process by introducing a formal solid model, and from which other geometric properties of rigid solid objects can be reasoned about.

The difficulty in the present application is the verification problem when designing mechanical parts of transformers with complicated shapes to fit into a certain predefined environmental space. Furthermore, the electrical characteristics of the transformer itself are heavily dependent upon the relative spatial positions and dimensions of their components such as core, coils and cooling sections [3,38]. This means that a valid geometric model will play a key role in the verification of the transformer design process [1].

However, the need for a verified modelling theory has to be preceded by a thorough understanding of what does constitutes a valid, complete and unique model, and this motivation results in the work described in this chapter. In the following sections, a formal geometric modelling framework which is based on Higher-Order Logic and set-theoretic topology is introduced. This chapter describes the work involving in modelling rigid solid objects as set-theoretic compositions of primitive solid building blocks. The solid geometry is based on Euclidean geometry and general point-set topology [102]. The mathematical foundations and certain elementary, but specialised results, of point-set topology which are required in the study of geometric modelling are also discussed.

The development of the geometric modelling theory is in three presented stages. The first part describes the mathematical foundation of point-set topology which is represented and verified using Higher-Order logic. The second part gives a formal representation of solid primitives and a Boolean model as its connective mechanism and, the last part discusses some practical examples of the application of theory to the specifications of the mechanical parts of a simple transformer.

7.2 Designing Mechanical Parts

In conventional design and manufacturing, information about dimensional (3-D) shapes is described using engineering drawings. However, drawing-based design presents several problems. This section discusses the problems involved in conventional 3-D design of mechanical parts, particularly to those of transformers, and then describes how geometric model-based design can solve these problems.

[1]This work was carried out in collaboration of Goodyear Transformers Ltd. in Birmingham.

7.2.1 Communication between the designer and the modeller

Although the designed object is 3-D, the information in the drawing is 2-D, so the information passed from the designer to the modeller is incomplete. Generally, the designer is not satisfied with the first model. When the shape includes many surfaces, the problem is particularly serious. To define these surfaces, therefore, a designer will draw many cross-sections. However, the designer often cannot adequately express the shape that he/she envisages, so the first model (or drawing) may need many modifications. To satisfy these design requirements, the modeller will then make a new model and this process will be repeated as necessary. However, because both the finance and the time available for design are in general limited, the designer is not always satisfied with the final model.

7.2.2 Time taken and costs incurred in the design process

After the designer has specified the modifications required, obviously he/she would like the new model to be made as quickly as possible, and for the modeller to make as many models as necessary. However, it takes some time to make each model, and they are generally expensive to produce. In developing a commercial product, it is desirable to shorten the time taken in designing it and to minimise expenditure. It is therefore difficult to allow the designer to produce as many models as he or she might wish.

To solve these problems, a system of design based on geometric modelling can be envisaged. Fig.7.1*b* shows such a design process. In this method, the designer does not produce a finished drawing, but works interactively on a computer to refine the design and produce a geometric model. To ensure the validity and rigidity of the design, the model is manipulated using a number of correctness-preserving transformations. That is each geometrical operation applied to the valid model should produce a new single or compound solid that is also valid.

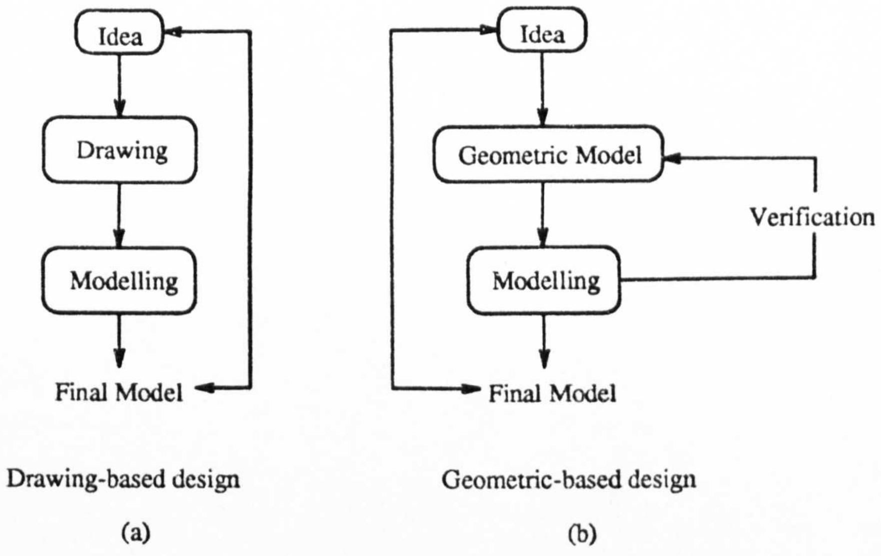


Figure 7.1: The use of formal geometrical model

7.3 Geometric Modelling in Transformer Design

A geometric modelling theory does not only provide solutions to the communication problems outlined above but it also supports the following geometric verifications which are crucial in the development of high-cost products. This present study will concentrate on one such example which is the design of large power transformers used in safety-critical applications such as nuclear power plans, or in a space-limited environment such as the Channel Tunnel.

7.3.1 Spatial arrangement

A typical schematic diagram of a power transformer is shown in Fig.7.2 with its usual fittings. An important consideration in designing transformers is that the electrical characteristics are closely related to its mechanical arrangement. For example, in the design of winding coils and former, the designer has to make sure that the windings will fit into the window space of the laminated core. A small change in the dimensions of the former or coils will greatly affect the reactance paths and consequently the eddy losses of the transformer. Moreover, additional fittings such as bushings, switch gears and tap changers, further complicate the mechanical design.

7.3.2 Volumetric verification

When transformers are designed for indoor applications, the spatial occupancy becomes more important. Since large power transformers normally have very complicated shapes, the task of keeping track of its spatial occupancy is not trivial, particularly when the design has a number of modifications, which is the usual case. Therefore, by presenting the mechanical structure of the design using a formal solid model the overall space of the finished product can then be deduced.

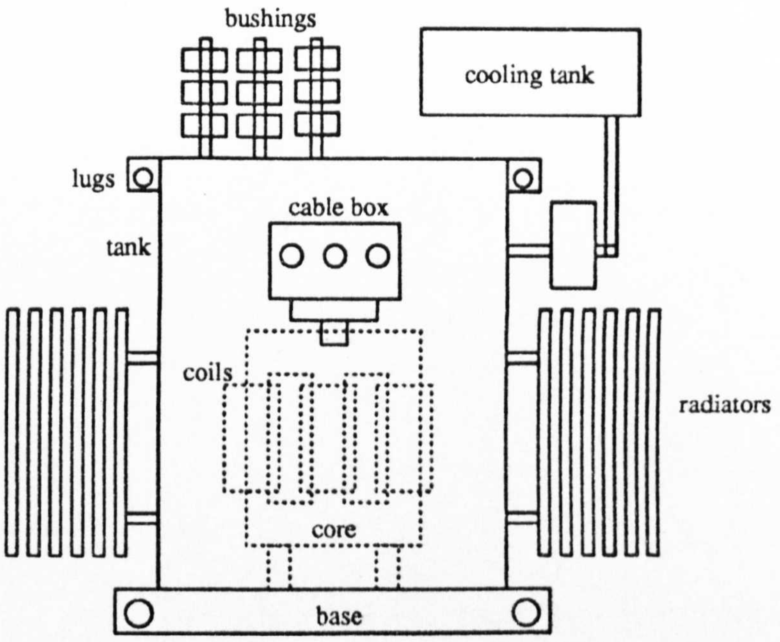
To represent the compositional relationship between the various mechanical parts, we use a formal model of a binary tree [2] to verify geometric operations and the design's verification is achieved indirectly by ensuring that each operation in the resultant btree is a formally valid transformation. The approach in this case study can easily be justified. Let us consider the compositional representation of the btree of the simplified transformer shown in Fig.7.2. It is obvious that even with a very simple compound object, the verification of its validity is not a trivial task, and furthermore the size of the btree grows rapidly as the model becomes more complex. Thus, if solid primitives and their associated operations can be formally represented as geometrically valid transformations then the validity of the final object can be implicitly assured.

The following sections discuss some representational issues and requirements of a valid solid modelling system, and then from these mathematical foundations its formal specification can be defined and then verified using appropriate topologies.

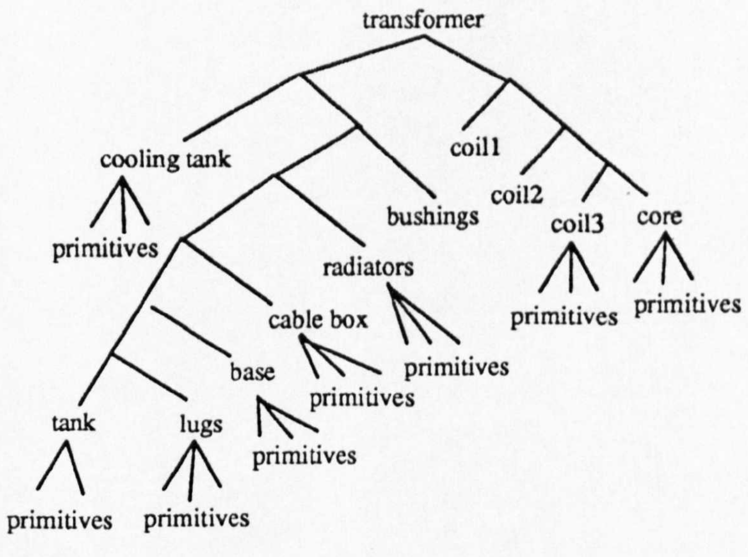
7.4 Mathematical Foundations

There are many different viewpoints about the notions of representational completeness and consistency. A definition of the so-called geometrical completeness

[2]This is also referred to as a btree.



(a) An oversimplified power transformer diagram



(b) Structure of the compositional tree

Figure 7.2: The schematic diagram of a simplified transformer

can be described as follows [10],

A specification is geometrically complete if it contains information sufficient to compute and reason about the geometric properties of the specified entity.

Fig.7.3 suggests a useful way of looking at the problem. Firstly objects in the real world must be replaced by a mathematical model, ie. by abstract entities defined in an appropriate domain of mathematics. This domain includes subsets of the three-dimensional Euclidean space which possess the properties of compactness and regularity. This class of subsets excludes curves, surfaces, and other sets which are not solid, ie. homogeneously three dimensional or simply regular.

The geometric property can be precisely defined by the mappings or functions which take regular sets and produce well-defined mathematical formulae. For instance, volume is a geometrical property of an object because it can be defined by a function that maps regular sets of points into positive numbers.

The relation labelled representation scheme in Fig.7.3 takes regular sets and produces mathematically defined symbol structures. Eventually such symbol structures must be further mapped into storage structures in the computer but this step is not considered here. The notion of geometrical completeness may be expressed quite simply in this framework : it requires that the inverse of the representation scheme relation be a single-valued function, i.e. that each representation corresponds to a one-one function f which satisfies the relation,

$$\vdash \forall f.\, \mathsf{ONE\_ONE}\, f = \forall x_1\, x_2.\, (f\, x_1 = f\, x_2) \supset (x1 = x2)$$

Clearly, if the set that corresponds to a given representation is known then we have all the geometrical information which is required to determine its geometric properties.

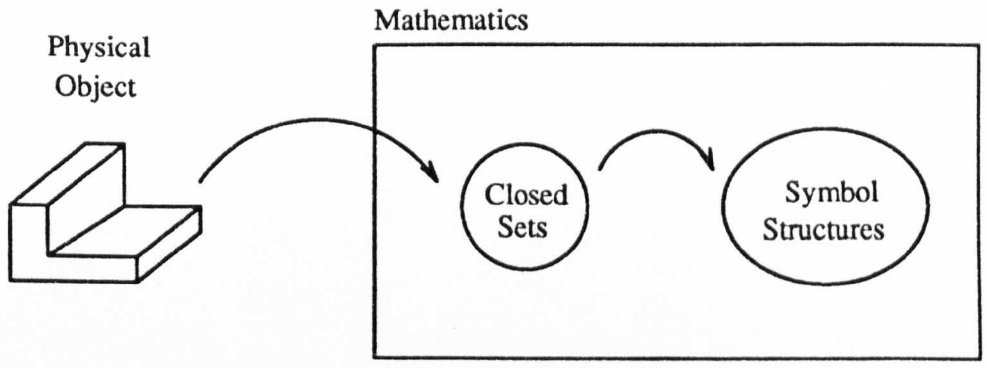


Figure 7.3: Representation scheme

7.5 Point-Set Algebra

In solid modelling, when we combine simple shapes, called primitives, to form more complex shapes, the application of set theory becomes useful. Solid modelling techniques, in particular have drawn considerably from the axioms of set theory. The term set denotes a well-defined collection of objects. Objects belonging to a set are its elements or members. The basic element in our modelling approach is the point. The application of set operations to sets of points in Cartesian space is therefore referred to as point-set algebra.

It should be noted that definitions and theorems represented in this chapter are using the general form of set notation, that is, a set S which contains all points x that satisfies a relation $P(x)$ is denoted as,

$$S = \{x \mid P(x)\}$$

where the symbol '|' is called a set-builder notation and it describes the set in terms of conditions on any arbitrary element of the set. For example,

$$S = \{x \mid 1 < x < 10\}$$

is a set consisting of all natural numbers in the specified interval excluding its end points. The condition on the right hand side of the vertical line represents the condition for set membership.

Any set that contains all the elements of all the sets under consideration is the universal set UNIV, and conversely, EMPTY is a set which has no elements. More formally,

$\vdash \mathsf{UNIV} = \{x \mid \mathsf{true}\}$

$\vdash \mathsf{EMPTY} = \{x \mid \mathsf{false}\}$

There are three basic operations which are frequently used in this modelling theory. These are intersection, union, and difference between sets. They are defined in HOL as follows,

$\vdash \forall s\, t.\, s\ \mathsf{UNION}\ t = \{x \mid x \in s \lor x \in t\}$

$\vdash \forall s\, t.\, s\ \mathsf{INTER}\ t = \{x \mid x \in s \land x \in t\}$

$\vdash \forall s\, t.\, s\ \mathsf{DIFF}\ t = \{x \mid x \in s \land \neg x \in t\}$

The complement of s with respect to the universal set UNIV is denoted by COMP s,

$\vdash \forall s.\, \mathsf{COMP}\ s = \mathsf{UNIV}\ \mathsf{DIFF}\ s$

The following theorems about basic properties of set topology [71] can easily be proved [3],

$\vdash \forall s.\, s\ \mathsf{UNION}\ (\mathsf{COMP}\ s) = \mathsf{UNIV}$

$\vdash \forall s.\, s\ \mathsf{INTER}\ (\mathsf{COMP}\ s) = \mathsf{EMPTY}$

$\vdash \forall s\, t.\, \mathsf{COMP}\ (s\ \mathsf{UNION}\ t) = (\mathsf{COMP}\ s)\ \mathsf{INTER}\ (\mathsf{COMP}\ t)$

$\vdash \forall s\, t.\, \mathsf{COMP}\ (s\ \mathsf{INTER}\ t) = (\mathsf{COMP}\ s)\ \mathsf{UNION}\ (\mathsf{COMP}\ t)$

$\vdash \forall s\, t.\, \mathsf{COMP}\ (s\ \mathsf{DIFF}\ t) = (\mathsf{COMP}\ s)\ \mathsf{UNION}\ t$

$\vdash \forall s\, t.\, (s = t) \supset (\mathsf{COMP}\ s = \mathsf{COMP}\ t)$

$\vdash \forall s.\, \mathsf{COMP}(\mathsf{COMP}\ s) = s$

[3]The proofs are included in Annex D.

A comprehensive list of theorems about the basic properties of sets is included in Annex D [4]. The following sections describe some elementary, but essential concepts which are used in the geometric modelling theory.

7.5.1 Metric space

A metric space is a pair (s, d) where s is a set and $d : * \times * \rightarrow num$ is a function called the *distance* or the *metric* function, such that for all x, y and z in s, the function d asserts the following properties,

$$\begin{aligned}
&\vdash \forall d. \mathsf{METRIC}\, d = \\
&\quad (\forall x\, y. 0 \leq d\, x\, y) \wedge \\
&\quad (\forall x\, y. d\, x\, y = d\, y\, x) \wedge \\
&\quad (\forall x\, y. x = y \supset d\, x\, y = 0) \wedge \\
&\quad (\forall x\, y\, z. d\, x\, z \leq d\, x\, y + d\, y\, z)
\end{aligned}$$

We may think of the distance function d as providing a quantitative measure of the degree of closeness of two points. In particular, the triangularity relation

$$\vdash \forall x\, y\, z. d\, x\, z \leq d\, x\, y + d\, y\, z$$

may be thought of as an assertion of the transitivity of closeness; that is, if x is close to y and y is close to z, then x is close to z. The existence theorem follows immediately from the definition of the METRIC definition,

$$\vdash \forall x. \exists R\, d. x \in s \wedge \mathsf{METRIC}\, d \supset (d\, x\, x) \leq R$$

This means that for any point x in a set s, there always exists an appropriate function to qualify the set s to be a metric space. However, it should be noted that the metric function d represents an abstract measure of the degree of closeness of pairs of points and it does not necessarily relate to the concept of the usual

[4]Theorems about algebraic topology were originally introduced by Artin [6].

geometrical distance. For instance, suppose that a metric space is the Euclidean geometry with its concept of distance defined by a metric function as follows,

$$\vdash \forall x\ y.\,\mathsf{DISTANCE}\,(x,y) = (x = y)\ \Rightarrow\ 0\ |\ 1$$

then it can easily be proved that the set of all points defined by DISTANCE satisfy the metric space [5], that is

$$\vdash \mathsf{METRIC}(\mathsf{DISTANCE})$$

In fact, DISTANCE describes a 2-dimensional grid of points in the first quadrant of the Cartesian coordinates. In fact, an integer is defined in HOL as a tuple of pair (b,n), where n is a natural number and b is a boolean value which is true if n represents a positive integer, and false otherwise. The first quadrant of Cartesian coordinate is therefore represented as a 2-tuple of pair ((true,x),(true,y)), where x and y are coordinates of integer values.

In geometry, we are usually concerned with sets that have no limit points. For example, the set of all points in the Cartesian space is infinite and one often refers to this class of sets as open sets [6].

7.5.2 Open sets and neighbourhood

A subset s of a metric space is open if it contains an open ball about each of its points, that is

$$\vdash \forall s.\,\mathsf{OPEN}s = \forall x.\,\exists R.\,x \in \mathsf{OPEN\_BALL}\ s\ x\ R$$

where the circle about each point x in s is defined by the relation OPEN_BALL, where

$$\vdash \forall s\ x\ R.\,\mathsf{OPEN\_BALL}\ s\ x\ R =$$
$$\{y \mid \exists d.\,(y \in s \land \mathsf{METRIC}\,d) \supset (d\ x\ y) \leq R\}$$

[5]The proof of this example is given in Annex D

This means that if the pair (s, d) is a metric space, and x is a point of s then the open ball of radius R about x, is the set of all points y whose distance from x is less than R. For instance, suppose s is a real line with its usual distance. An open ball about a real point x is an open interval of length $2R$ about x.

Open sets in a metric space have the following properties; the empty set EMPTY and the UNIV are open [6],

$\vdash$ OPEN (EMPTY : $(*)set$)

$\vdash$ OPEN (UNIV : $(*)set$)

The intersection of a finite number of open sets is an open set,

$\vdash \forall s\ t.$ OPEN s $\wedge$ OPEN t $\supset$ OPEN (s INTER t)

and, similarly, the union of a finite number of open sets is also an open set,

$\vdash \forall s\ t.$ OPEN s $\wedge$ OPEN t $\supset$ OPEN (s UNION t)

The neighbourhood of a point x in set s is any subset of s which contains an open set which contains x. Formally,

$\vdash \forall s\ x.$ NEIGHBOUR $(x,s) = \{y \mid (y \in s) \wedge (\exists y'. y' = x)\}$

A neighbourhood of x in s may be thought of as a set which contains all the points of s that are sufficiently close to x or *enclosing* x since it contains some OPEN_BALL about x. In particular, these open balls have the property that they are neighbourhood of each of their points, that is

$\vdash \forall s\ R\ y. y \in$ OPEN_BALL $(s, x, R) \supset$
  OPEN_BALL (s, x, R) = NEIGHBOUR (y, OPEN_BALL(s, x, R))

[6] Notation $(*)set$ represents sets of polymorphic type.

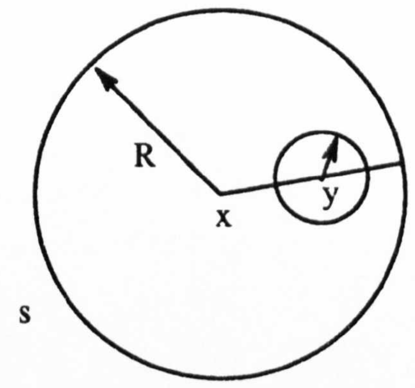


Figure 7.4: Neighbourhood in an open ball

The proof of this theorem can be illustrated pictorially in Fig.7.4 [7].

7.5.3 Closed sets

A subset s of a topological space is closed iff its complement is open,

$$\vdash \forall s.\,\mathsf{CLOSED}\ s = \mathsf{OPEN}\ (\mathsf{COMP}\ s)$$

It should be noted that closed sets are not the opposite of open sets. In fact, there are sets which are both open and closed, such as the UNIV and EMPTY set,

$$\vdash \mathsf{CLOSED}\ (\mathsf{UNIV} : (*)set) \tag{7.1}$$

$$\vdash \mathsf{CLOSED}\ (\mathsf{EMPTY} : (*)set) \tag{7.2}$$

The closed sets have the following properties,

$$\vdash \forall s\ t.\,\mathsf{CLOSED}\ s\ \wedge\ \mathsf{CLOSED}\ t\ \supset\ \mathsf{CLOSED}\ (s\ \mathsf{INTER}\ t) \tag{7.3}$$

$$\vdash \forall s\ t.\,\mathsf{CLOSED}\ s\ \wedge\ \mathsf{CLOSED}\ t\ \supset\ \mathsf{CLOSED}\ (s\ \mathsf{UNION}\ t) \tag{7.4}$$

It is clear that a set s is characterised by two subsets; one includes its interior members and the other contains those on its edges as illustrated in Fig.7.5 and Fig.7.6. It should be noted that the abstract concepts of OPEN and CLOSED may

[7]HOL proof of this theorem is included in Annex D.

lead to the incorrect conclusion that a set can not both be opened and closed. An obvious example is that, in any metric space, the two sets EMPTY and UNIV are open and consequently, their complements UNIV and EMPTY are both closed as shown in previous sections. There are also other subsets that are simultaneously open and closed, and they represent an important property of topology which is described as *connectedness*, that is

$$\vdash \forall s. \mathsf{CONNECTED}\ s = \mathsf{OPEN}\ s\ \wedge\ \mathsf{CLOSED}\ s$$

The concept of connectedness is useful in the formal definition of rigid solid objects as was outlined in section 7.6.1.

7.5.4 Closure

A point x is a limit point of a subset s of a topological space if each neighbourhood of x contains at least a point of s different from x.

$$\begin{aligned}&\vdash \forall s\ x. \mathsf{LIMIT}\ (x,s) = \\ &\qquad \forall s_1.(s_1 = \mathsf{NEIGHBOUR}(x,s))\ \wedge\ (\exists y. y\ \in\ s_1\ \wedge\ \neg\,(y = x))\end{aligned}$$

It can easily be proved that a set is closed iff it contains all its limit points,

$$\vdash \forall s. \mathsf{CLOSED}\ s = \forall x. \mathsf{LIMIT}\ (x,s)\ \supset\ (x\ \in\ s)$$

The closure of a subset s is then defined as the union of s with the set of all its limit points,

$$\vdash \forall s. \mathsf{CLOSURE}\ s = s\ \mathsf{UNION}\ \{x \mid \mathsf{LIMIT}\ (x,s)\}$$

It follows that if x is a point in the closure of s then each neighbourhood of x intersects s. In other words, the intersection of s with each neighbourhood of x results in non-empty sets,

$$\vdash \forall x\ s. x\ \in\ \mathsf{CLOSURE}\ s\ \supset\ \neg(s\ \mathsf{INTER}\ \mathsf{NEIGHBOUR}\ (x,s) = \mathsf{EMPTY})$$

Furthermore, a set s is closed iff it is identical to its closure, i.e.

$$\vdash \forall s. (\mathsf{CLOSED}\ s) = (s = \mathsf{CLOSURE}\ s) \tag{7.5}$$

The following theorems are proved for the properties of closure operations [8],

$$\vdash \forall s\, t. s \subset t \supset (\mathsf{CLOSURE}\ s) \subset (\mathsf{CLOSURE}\ t) \tag{7.6}$$

$$\vdash \forall s\, t. \mathsf{CLOSURE}\ (s\ \mathsf{UNION}\ t) = (\mathsf{CLOSURE}\ s)\ \mathsf{UNION}\ (\mathsf{CLOSURE}\ t)$$

$$\vdash \forall s\, t. \mathsf{CLOSURE}\ (s\ \mathsf{INTER}\ t) \subset (\mathsf{CLOSURE}\ s)\ \mathsf{INTER}\ (\mathsf{CLOSURE}\ t) \tag{7.7}$$

The interesting fact about the last property (7.7) is that the closure of set intersection may be strictly smaller than the set intersection itself. For instance, let us suppose that s and t are the intervals (0,5) and (5,7), then

$$\mathsf{CLOSURE}\ (s\ \mathsf{INTER}\ t) = \{\}$$

whilst the intersection of their closures is a singleton set,

$$(\mathsf{CLOSURE}\ s)\ \mathsf{INTER}\ (\mathsf{CLOSURE}\ t) = \{5\}$$

It should be noted from theorem (7.5) that by taking the CLOSURE of a set, we apparently associate it to a new subset, denoted by CLOSURE s. This correspondence or operation on sets satisfies the following properties whose proofs are given in Annex D,

$$\vdash \mathsf{CLOSURE}\ (\mathsf{EMPTY}) = \mathsf{EMPTY}$$

$$\vdash \forall s. \mathsf{CLOSURE}\ s = s$$

$$\vdash \forall s. s \subset (\mathsf{CLOSURE}\ s)$$

$$\vdash \forall s\, t. \mathsf{CLOSURE}\ (s\ \mathsf{UNION}\ t) = \mathsf{CLOSURE}\ s\ \mathsf{UNION}\ \mathsf{CLOSURE}\ t$$

$$\vdash \forall s. \mathsf{CLOSURE}\ (\mathsf{CLOSURE}\ s) = \mathsf{CLOSURE}\ s$$

[8] Derivations of proofs are shown in Annex D.

These properties (or more precisely, theorems) are considered as a set of axioms for what we will call a *closure space* and then from theorems (7.1-7.6) it follows that there is a one-one correspondence between the topological spaces and the closure spaces.

So far, it has been shown that the closure of a set s is the smallest closed subset containing s. Another significant subset associated with s is the interior of s, which is the largest open set contained in s.

7.5.5 Interior

An interior of a subset s of a topological space is a set which contains all points x such that there is at least a neighbourhood of x which is an open subset of s. More formally,

$$\vdash \forall s.\,\mathsf{INTERIOR}\ s = \{x \mid \exists t.\,\mathsf{OPEN}\ t \wedge (t \subset s) \wedge (x \in t)\}$$

Consequently, a point x is inside a set s iff it belongs to the interior of s,

$$\vdash \forall s\ x.\,\mathsf{INSIDE}\,(x,s) = x \in \mathsf{INTERIOR}\ s$$

It follows immediately that if x is inside s then its neighbourhood is equal to s,

$$\vdash \forall x\ s.\,\mathsf{INSIDE}\,(x,s) \supset (s = \mathsf{NEIGHBOUR}(x,s))$$

It is clear that $\mathsf{INTERIOR}\ s$, being the union of open sets, is itself open [9], and it is the largest open set contained in s, that is

$$\vdash \forall s.\,(s = \mathsf{INTERIOR}\ s) = \mathsf{OPEN}\ s$$

Furthermore, if s contains a family of open sets then the complement of its interior is the family of closed sets contains $\mathsf{COMP}\ s$,

$$\vdash \forall s.\,\mathsf{COMP}\ (\mathsf{INTERIOR}\ s) = \mathsf{CLOSURE}(\mathsf{COMP}\ s)$$

[9] From section 7.5.2

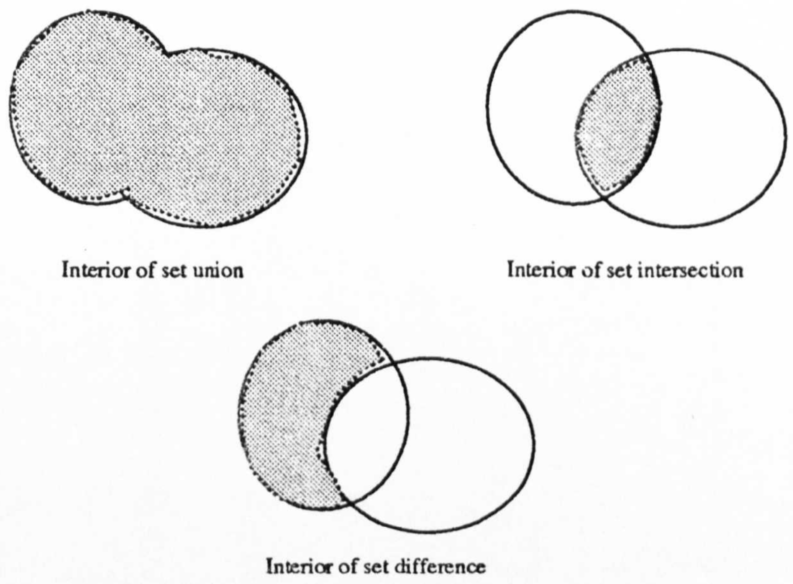


Figure 7.5: Defining the interior of set operations

and, conversely

$$\vdash \forall s.\ \mathsf{INTERIOR}\ s = \mathsf{COMP}(\mathsf{CLOSURE}(\mathsf{COMP}\ s))$$

The interior of a set which results from set operations such as intersection, union and difference is illustrated in Fig.7.5 and the following theorems have been proved using the HOL system,

$$\vdash \forall s\ t.\ \mathsf{INTERIOR}\ (s\ \mathsf{UNION}\ t) = (\mathsf{INTERIOR}\ s)\ \mathsf{UNION}\ (\mathsf{INTERIOR}\ t)$$

$$\vdash \forall s\ t.\ \mathsf{INTERIOR}\ (s\ \mathsf{INTER}\ t) = (\mathsf{INTERIOR}\ s)\ \mathsf{INTER}\ (\mathsf{INTERIOR}\ t)$$

$$\vdash \forall s\ t.\ \mathsf{INTERIOR}\ (s\ \mathsf{DIFF}\ t) = (\mathsf{INTERIOR}\ s)\ \mathsf{DIFF}\ (\mathsf{INTERIOR}\ t)$$

Further theorems about the interior operation on sets are included in Annex D.

7.5.6 Boundary

A boundary of a subset s is a set which includes all points x such that x belongs to both s and its complement set, thus

$$\vdash \forall s.\ \mathsf{BOUNDARY}\ s =$$

$$\{x \mid (x \in \mathsf{CLOSURE}\,s) \wedge (x \in \mathsf{CLOSURE}\,(\mathsf{COMP}\,s))\} \qquad (7.8)$$

Consequently, a point x is considered to be on the boundary of a subset s iff it is a member of the boundary subset,

$$\vdash \forall s\, x.\, \mathsf{ON}\,(x,s) = x \in \mathsf{BOUNDARY}\,s$$

Definition (7.8) asserts informally that the boundary of a set s contains all points that are arbitrarily close to both s and its complement set, that is

$$\vdash \forall s.\, \mathsf{BOUNDARY}\,s = (\mathsf{CLOSURE}\,s)\ \mathsf{INTER}\ \mathsf{CLOSURE}\,(\mathsf{COMP}\,s)$$

It follows that s and COMP s have the same boundary, since

$$\begin{aligned}&\vdash \forall s.\, \mathsf{BOUNDARY}\,(\mathsf{COMP}\,s) = \\ &\qquad \mathsf{CLOSURE}\,(\mathsf{COMP}\,s)\ \mathsf{INTER}\ \mathsf{CLOSURE}\,(\mathsf{COMP}(\mathsf{COMP}\,s))\end{aligned} \qquad (7.9)$$

and it was shown earlier that,

$$\vdash \forall s.\, \mathsf{COMP}(\mathsf{COMP}\,s) = s \qquad (7.10)$$

Hence, by rewriting theorem (7.9) with theorem (7.10) we have,

$$\vdash \forall s.\, \mathsf{BOUNDARY}\,(\mathsf{COMP}\,s) = \mathsf{CLOSURE}\,(\mathsf{COMP}\,s)\ \mathsf{INTER}\ \mathsf{CLOSURE}\,s$$

It follows immediately from the symmetrical property of the INTER operation and the definition of BOUNDARY that,

$$\begin{aligned}&\vdash \forall s.\, \mathsf{BOUNDARY}\,(\mathsf{COMP}\,s) \\ &\qquad = (\mathsf{CLOSURE}\,s)\ \mathsf{INTER}\ \mathsf{CLOSURE}\,(\mathsf{COMP}\,s) \\ &\qquad = \mathsf{BOUNDARY}\,s\end{aligned} \qquad (7.11)$$

Furthermore, for any set s in a topological space, it can easily be proved that,

$\vdash \forall s.\,\mathsf{CLOSED}(\mathsf{BOUNDARY}\ s)$

$\vdash \forall s.\,\mathsf{CLOSURE}\ s = (\mathsf{INTERIOR}\ s)\ \mathsf{UNION}\ (\mathsf{BOUNDARY}\ s)$

$\vdash \forall s.\,\mathsf{CLOSURE}\ s = (s\ \mathsf{UNION}\ \mathsf{BOUNDARY}\ s)$

$\vdash \forall s.\,(\mathsf{INTERIOR}\ s)\ \mathsf{DISJOINT}\ (\mathsf{BOUNDARY}\ s)$

Up to this point, we have formally shown that in any topological space, a closed set can be uniquely defined by its interior subset and a boundary subset. Any operations on sets should therefore preserve the closed-set property. This means that they also result in a valid set with a uniquely defined interior and boundary subsets. It was shown in section 7.5.5 and illustrated in Fig.7.5 that the interior of set operations on any two sets is a proper subset. However, it is more difficult to define the boundary of the resultant set.

Let us consider a simple example of defining the boundary of set intersection as shown in Fig.7.6. The boundary of s INTER t comprises of three subsets, denoted by A, B, and C. The problem of defining the boundary of s INTER t reduces to a simpler problem of identifying these three subsets. It can be shown geometrically from Fig.7.6, and is shown formally in Annex D, that the subset A satisfies the relation,

$\vdash A = (\mathsf{BOUNDARY}\ t)\ \mathsf{INTER}\ (\mathsf{INTERIOR}\ s)$

and, similarly

$\vdash B = (\mathsf{INTERIOR}\ t)\ \mathsf{INTER}\ (\mathsf{BOUNDARY}\ s)$

It is obvious that the subset C includes the isolated points at the boundary junctions, i.e.

$\vdash C = (\mathsf{BOUNDARY}\ s)\ \mathsf{INTER}\ (\mathsf{BOUNDARY}\ t)$

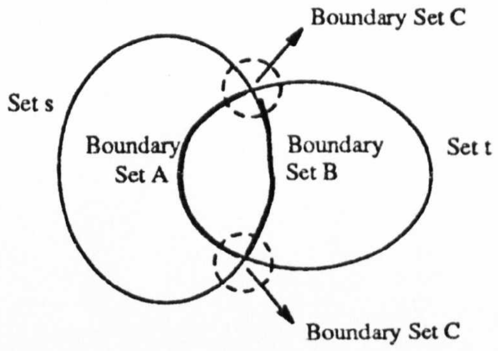


Figure 7.6: Set compositions of the boundary of s INTER t

Furthermore, to account for the *gaps* around these boundary junctions, we have to include the intersection of both closures of s and t, the relation for defining subset C then becomes,

$\vdash C =$ (BOUNDARY s) INTER (BOUNDARY t) INTER CLOSURE (s INTER t)

In fact, the following theorem has been verified in HOL and its proof is given in Annex D,

$\vdash \forall s\ t.$ BOUNDARY (s INTER t) =

(BOUNDARY t INTER INTERIOR s) UNION

(INTERIOR t INTER BOUNDARY s) UNION

(BOUNDARY s INTER BOUNDARY t INTER

CLOSURE (s INTER t)) (7.12)

The boundary of set unions and differences can be verified similarly, and theorems for defining their boundaries are given below,

$\vdash \forall s\ t.$ BOUNDARY (s UNION t) =

(BOUNDARY s INTER INTERIOR (COMP t)) UNION

(INTERIOR (COMP s) INTER BOUNDARY t) UNION

(BOUNDARY s INTER BOUNDARY t INTER

$$\text{CLOSURE (COMP } s \text{ INTER COMP } t)) \tag{7.13}$$

$$\vdash \forall s\, t.\, \text{BOUNDARY } (s \text{ DIFF } t) =$$
$$(\text{BOUNDARY } s \text{ INTER INTERIOR (COMP } t)) \text{ UNION}$$
$$(\text{INTERIOR } s \text{ INTER BOUNDARY } t) \text{ UNION}$$
$$(\text{BOUNDARY } s \text{ INTER BOUNDARY } t \text{ INTER}$$
$$\text{CLOSURE } (s \text{ INTER COMP } t)) \tag{7.14}$$

It follows immediately from theorems (7.12-7.14) that,

$$\vdash \forall s\, t.(s \subset t) \supset$$
$$t = (\text{INTERIOR } s \text{ UNION BOUNDARY } s \text{ UNION INTERIOR (COMP } s))$$
$$\vdash \forall s\, t.\, \text{BOUNDARY } (s \text{ UNION } t) \subset (\text{BOUNDARY } s \text{ UNION BOUNDARY } t)$$
$$\vdash \forall s\, t.\, \text{BOUNDARY } (s \text{ INTER } t) \subset (\text{BOUNDARY } s \text{ UNION BOUNDARY } t)$$
$$\vdash \forall s\, t.\, \text{BOUNDARY } (s \text{ DIFF } t) \subset (\text{BOUNDARY } s \text{ UNION BOUNDARY } t)$$
$$\vdash \forall s\, t.\, \text{INTERIOR } (s \text{ UNION } t) \subset (\text{INTERIOR } s \text{ UNION INTERIOR } t$$
$$\text{UNION (BOUNDARY } s \text{ INTER BOUNDARY } t))$$

7.5.7 Dimensional property of boundary

Let us consider a set s which includes all the points in the interval (0,1). It can easily be shown from the definitions described in earlier sections that,

$$\text{INTERIOR } s = \{\}$$
$$\text{CLOSURE } s = \{0, 1\}$$
$$\text{BOUNDARY } s = \{0, 1\}$$

What has been illustrated in this example is that, when the interior of a set is empty, the abstract concept of the boundary of a set s is a *thin* set which separates s from its complement counterpart COMP s is not adequately represented. Since, in this case,

$$\text{BOUNDARY } s = \text{CLOSURE } s$$

from which the BOUNDARY is interpreted as having some degree of thickness. Therefore in this section, we introduce a class of sets whose boundaries are suitably *thin* for the intuitive concept of BOUNDARY to be justifiable. A set s is considered to be thin if it satisfies the relation,

$$\vdash \forall s.\, \text{THIN } s = \text{INTERIOR } (\text{CLOSURE } s) = \{\}$$

It should be noted that THIN sets have empty INTERIOR but the converse is not true in general. For instance, the set of rationals has empty interior and yet it is not a THIN set, since the interior of its closure is the whole real line.

The following properties ensures that any operation on sets preserves the dimensionality of the resulting component, that is

$$\vdash \forall s\, t.\, \text{THIN } s \wedge \text{THIN } t \supset \text{THIN } (s \text{ UNION } t)$$
$$\vdash \forall s\, t.\, \text{THIN } s \wedge \text{THIN } t \supset \text{THIN } (s \text{ INTER } t)$$
$$\vdash \forall s\, t.\, \text{THIN } s \wedge \text{THIN } t \supset \text{THIN } (s \text{ DIFF } t)$$
$$\vdash \forall s\, t.\, (\text{THIN } s \wedge t \subset s) \supset \text{THIN } t$$

It follows that the abstract concept of a thin boundary can be defined formally in HOL as follows,

$$\vdash \forall s.\, \text{THINBOUND } s = \text{THIN } (\text{BOUNDARY } s)$$

and it can easily be verified that the boundary of any closed or open set is a thin set, that is

$$\vdash \forall s.\, \text{CLOSED } s \supset \text{THINBOUND } s \qquad (7.15)$$
$$\vdash \forall s.\, \text{OPEN } s \supset \text{THINBOUND } s \qquad (7.16)$$

The proofs of these theorems are shown in Annex D. The last property (7.16) can be readily be derived from (7.15) and theorem (7.11), which is shown in section 7.5.6 as follows; if s is open then from theorem (7.15) we can assert that,

$\vdash \forall s.$ THINBOUND (COMP s)

since COMP s is closed. Then from theorem (7.11), which states that a set s and its complement have the same boundary subset, that is

$\vdash \forall s.$ BOUNDARY (COMP s) = BOUNDARY s

It follows immediately that,

$\vdash$ THINBOUND (COMP s)
= THIN (BOUNDARY (COMP s))
= THIN (BOUNDARY s)
= THINBOUND s

7.6 Closed Point-Set Topology

So far, we have introduced the operations on open and closed sets which are of polymorphic types [10]. This means that these sets are defined in HOL to have an arbitrary type of the form $(*)set$ which represents a set taking all elements of the same $(*)$ type. In this modelling theory, we only deal with spatial properties and the set theory is therefore instantiated using an appropriate type definition. In this model the basic element is the point which is represented as a tuple of three elements,

point : $num \# num \# num$

[10] Definitions and properties of polymorphic types are discussed in Chapter 2.

This definition corresponds to a HOL type of three integers representing X, Y, Z coordinates respectively [11], where

$$\vdash \forall p. \mathsf{X}\ (p : \hat{}point) = \mathsf{FST}\ p$$

$$\vdash \forall p. \mathsf{Y}\ (p : \hat{}point) = \mathsf{FST}\ (\mathsf{SND}\ p)$$

$$\vdash \forall p. \mathsf{Z}\ (p : \hat{}point) = \mathsf{SND}\ (\mathsf{SND}\ p)$$

If we denote the three-dimensional Cartesian space as E^3 then any region R which is a subset of E^3 is a finite and bounded portion of E^3. Points that comprise any region are individually characterised as either lying entirely within the region or on its boundary. Thus the set of points denoted by R can be divided conveniently into two subsets $\mathsf{i}\ R$ and $\mathsf{b}\ R$, where $\mathsf{i}\ R$ is the set of points in the interior of a region and $\mathsf{b}\ R$ is the set of all points which are on its boundary. Hence,

$$\vdash \forall R.\ R = \mathsf{i}\ R\ \mathsf{UNION}\ \mathsf{b}\ R$$

where b and i are defined by the function BOUNDARY and INTERIOR [12], which maps the set of points R into its boundary and interior subset respectively. It follows that the BOUNDARY function has the following type,

$$\mathsf{BOUNDARY}\ :\ (point)set\ \#\ (point)set\ \rightarrow\ (point)set$$

and similarly,

$$\mathsf{INTERIOR}\ :\ (point)set\ \#\ (point)set\ \rightarrow\ (point)set$$

In general, a region can be described as a topological space which is defined by a tuple $(\mathsf{b}\ R, \mathsf{i}\ R)$ where $\mathsf{b}\ R$ and $\mathsf{i}\ R$ are the boundary and interior subsets of R respectively.

It follows that the closure of a region R, denoted CLOSURE R, is the union of

[11]The use of the caret ˆ symbol to represent polymorphic types is explained in Chapter 4.

[12]These functions are defined in section 7.5.5 and 7.5.6.

all its interior and boundary points, that is

$$\vdash \forall R.\,\mathsf{CLOSURE}\ R = \mathsf{b}\ R\ \mathsf{UNION}\ \mathsf{i}\ R$$

7.6.1 Defining OBJECT

To establish formal modelling criteria, we abstract a set of geometric characteristics from solid physical objects. The most important property is the concept of an object and it is determined by the total spatial point-set that defines it.

However, in geometric modelling, we are only interested in those objects that are bounded and connected. From the set topology discussed earlier in section 7.5.4, it follows that a set is bounded iff it is definable within a finite space that is if it is a closed set. Also from section 7.5.3 it has been shown that a set is connected iff it is both open and closed. An object must also satisfy certain conditions,

- It has distinguished member sets, ie. the boundary and interior are disjoint,
- It must be closed, ie. the boundary is a finite point-set, and
- Its closure is upper and lower bounded.

Thus, an object can be described formally in HOL as follows,

$$\begin{aligned}&\vdash \forall a.\,\mathsf{OBJECT}\ a = \mathsf{CONNECTED}\ a\ \wedge\\ &\qquad \neg(\mathsf{b}\ a = \mathsf{EMPTY})\ \wedge\ (\exists n.\,\mathsf{HAS\_CARD}\ (\mathsf{CLOSURE}\ a)\ n)\end{aligned}$$

where EMPTY represents the empty set {} and HAS_CARD is a binary function which returns a TRUE value if the cardinal of the set CLOSURE a is equal to n, and FALSE otherwise, that is

$$\vdash \forall s\ n.\,\mathsf{HAS\_CARD}\ s\ n = (\mathsf{CARD}\ s = n)$$

It should be noted that the definition is a partial specification and it is true under the intended interpretation. One important feature of the function OBJECT is

Class name	Boundary	Interior
Point	The point itself	No interior point
Curve	End points	Set of points on the curve except the end points
Surface	Finite closed curves	Set of points on surface except those on bounding curves
Solid	Finite surfaces	Set of points within solid except those on bounding surfaces

Table 7.1: Definitions of geometric objects

that it *specifies* an object rather than *describes* it. This means that it places bounds on certain aspects of its behaviour and remains silent on other aspects.

For example, if we assert that for an object a, OBJECT a delivers true, this asserts that it is an object within the domain of closed-set and remains silent about its characteristics outside that domain. In this context, it is interesting to note that a dot which has a single point in its boundary and an empty interior is also a valid object however, as it will be shown later in section 7.6.2 that it is not a valid solid primitive.

Other basic geometric objects can be defined similarly in HOL as classified in Table 7.1, from which, for example, a CURVE is defined as an object having an finite BOUNDARY set of points and a non-empty INTERIOR set as follows,

$$\vdash \forall c.\ \mathsf{CURVE}\ c = \mathsf{OBJECT}\ c\ \wedge\ \neg(\mathsf{i}\ c = \mathsf{EMPTY})\ \wedge\ \mathsf{FINITE}\,(\mathsf{b}\ c)$$

The definition can be expanded to include special cases for CLOSED_CURVE and OPEN_CURVE as follows,

$$\vdash \forall c.\ \mathsf{CLOSED\_CURVE}\ c = \mathsf{CURVE}\ c \wedge\ (\mathsf{b}\ c = \mathsf{EMPTY})$$

$$\vdash \forall c.\ \mathsf{OPEN\_CURVE}\ c = \mathsf{CURVE}\ c\ \wedge\ \mathsf{HAS\_CARD}\ (\mathsf{b}\ c)\ 2$$

A surface is then defined as having a BOUNDARY which contains one or more

CLOSED_CURVE and a finite INTERIOR set,

$$\begin{aligned}&\vdash \forall s.\,\mathsf{SURFACE}\ s = \mathsf{OBJECT}\ s\ \wedge\\ &\qquad \neg(\mathsf{i}\ s = \mathsf{EMPTY})\ \wedge\ (\exists c.\,\mathsf{CLOSED\_CURVE}\ c\ \wedge\ \mathsf{CLOSURE}\ c\ \subset\ (\mathsf{b}\ s))\end{aligned}$$

7.6.2 Solid primitives

As is shown in Table 7.1, a solid primitive must satisfy the following set of conditions,

- A finite number of surfaces defines the boundary of a solid
- A face of a solid is a subset of the solid's boundary
- The union of all faces of an object defines its boundary

Therefore, a solid primitive can be defined in HOL as follows,

$$\begin{aligned}&\vdash \forall p.\,\mathsf{PRIMITIVE}\ p = \mathsf{OBJECT}\ p\ \wedge\\ &\qquad (\exists s.\,\mathsf{SURFACE}\ s\ \wedge\ (\mathsf{CLOSURE}\ s\ \subset\ \mathsf{b}\ p)\ \wedge\ \neg(\mathsf{i}\ p = \mathsf{EMPTY}))\end{aligned}$$

It can be shown that three basic properties of geometric solids [20], ie. boundedness, connectedness and point inclusion can be derived as shown below.

A primitive comprises of a bounded subset of the Cartesian space if it is bounded, i.e. it has a finite CLOSURE subset. Formally, it can be proved that,

$$\vdash \forall p.\,\mathsf{PRIMITIVE}\ p\ \supset\ \mathsf{FINITE}\ (\mathsf{CLOSURE}\ p)$$

and a solid primitive is connected if any two points inside its interior can be joined by a line without exiting the interior.

$$\begin{aligned}&\vdash \forall p.\,\mathsf{PRIMITIVE}\ p\ \supset\\ &\qquad (\forall x_1\ x_2.(x_1\ \in\ p\ \wedge\ x_2\ \in\ p))\ \supset\\ &\qquad\qquad (\exists c.(\mathsf{CURVE}\ c\ \wedge\ c\ \subset\ p\ \wedge\ x_1\ \in\ c\ \wedge\ x_2\ \in\ c))\end{aligned}$$

$$\supset (\forall x'. x' \in c \supset x' \in p)$$

The theorem asserts that if p is a PRIMITIVE then for all points x_1 and x_2 inside p, there is always a CURVE c, which is a subset of p, that joins these two points without leaving the INTERIOR of p.

It follows immediately that, given a primitive solid p, the position of all points in the Cartesian space can be defined relative to p, that is

$$\vdash \forall p\, x. x \in (\mathrm{i}\, p) \vee \neg(x \in \mathsf{CLOSURE}\, p) \vee x \in (\mathrm{b}\, p)$$

Up to this point, we have formally defined and consequently verified basic properties of a rigid solid model, that is of an object which satisfies the relation PRIMITIVE. The question now is how to define quantitatively how such an object can be constructed in the real world. There are a number of modelling systems which use different methods of modelling solid objects. Three basic types of modelling systems are the wire-frame model, surface model and solid model [19,78,20,94].

In a wire-frame model, objects are represented by a table defining edges and points [78]. The start point and the end point of each edge are stored in the edge table. An edge may be a line or a curve. The coordinates (x,y,z) of each point are stored in the point table. This representation is simple and natural for designers who is familiar with mechanical drawings, since it is the lines and curves in a drawing that define a 3-D shape. However, a wire-frame model is ambiguous when determining the surface area and volume of an object since different interpretations can be derived from a particular wire-frame drawing [99].

A surface model is represented by a set of edges and points, as is a wire-frame model, plus a set of faces. The surface-model is unambiguous when determining the volume and other spatial properties of single object which is defined by unique surfaces. However, it is more difficult to represent a combination of a number of objects of different shapes and sizes [28].

A solid model defines an object as a spatial point set which serves consequently as an abstract definition of solids [20,94]. The object's representation is unambiguous and subsequently, the model can be used to reason about many basic properties of physical objects such as interference and membership classification. However, the main problem with solid model representation is that it lacks quantitative measures for physical characteristics such as the representation of volume and surface areas, and therefore, the solid model is often referred to as an *unevaluated* model [87].

Our approach in this modelling theory is a combination of solid representation and surface model. That is, a solid primitive is defined as an object which satisfies the PRIMITIVE predicate and its boundary is defined by boolean combinations of directed half-surfaces.

Informally, a directed surface is a surface whose normal at any point determines the inside and outside of the primitive solid. Since an unbounded surface divides the Cartesian space into two regions, each region is called a half-space, and the surface is therefore referred to as a half-surface. The region which is nearest to the origin of the coordinate is called a NEAR_SPACE and the other further away from the origin is a FAR_SPACE. The boolean intersection of an appropriate set of half-spaces can form a closed 3-D solid. An object, or primitive, can thus be defined by the union of the intersection of all its directed surfaces [80], as follows,

$$F = \overset{m}{\cup} \left(\overset{n}{\cap}\ f_{ij} \right)$$

where f_{ij} are directed surfaces or half-spaces, and the directed surfaces can be formally defined in HOL as follows,

$\vdash \forall f.\ \mathsf{FAR\_SPACE}\ f(k,l,m,n) = \mathsf{SURFACE}\ f\ \wedge$

$\quad (\forall x.(x \in \mathsf{CLOSURE}\ f) \supset (k * \mathsf{X}(x) + l * \mathsf{Y}(x) + m * \mathsf{Z}(x)) \geq n)$

$\vdash \forall f.\ \mathsf{NEAR\_SPACE}\ f(k,l,m,n) = \mathsf{SURFACE}\ f\ \wedge$

$\quad (\forall x.(x \in \mathsf{CLOSURE}\ f) \supset (k * \mathsf{X}(x) + l * \mathsf{Y}(x) + m * \mathsf{Z}(x)) \leq n)$

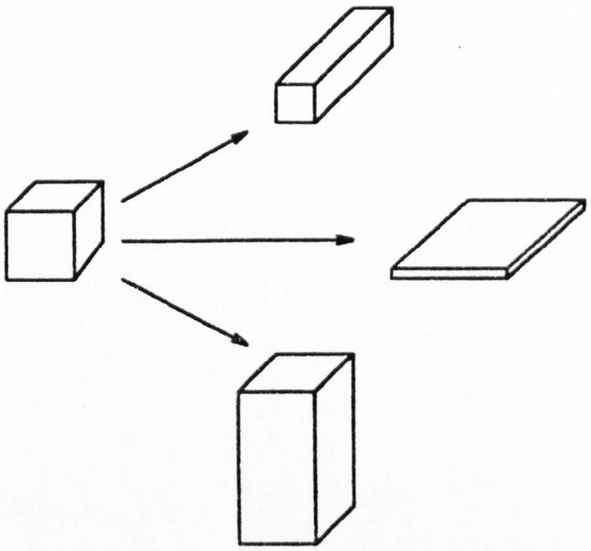


Figure 7.7: Scaling transformation

It should be noted that points in the boundary subset belong to both the NEAR_SPACE and FAR_SPACE surfaces, that is

$$\vdash \forall p\, x.\, x \in (\mathsf{b}\ p) \supset$$
$$(x \in \mathsf{NEAR\_SPACE}\ f(k,l,m,n) \wedge x \in \mathsf{FAR\_SPACE}\ f(k,l,m,n))$$

7.6.3 Generic and parameterised primitives

The approach taken in this modelling system is to generate a finite set of basic primitives whose size, shape and orientation are determined by a set of user-defined parameters. The primitives themselves are represented by the intersection of a set of curved or planar half-spaces. For example, the primitive block is represented by the intersection of six planar half-spaces and the cylinder by the intersection of a cylindrical half-space and two planar half-spaces.

A direct way of defining a new solid is to apply a sequence of simple linear transformations to an existing one. Consider a unit cube. There are several ways of transforming it into a new shape by using scaling operators. Fig.7.7 is an example of how an unlimited variety of specific instances of an original solid is created by simple scaling transformations. Notice that such transformations affect the geometry but not the topology of a shape, that is, the transformation preserves the shape of the modified objects [2].

To adequately define the shape of a relatively simple class of objects such as blocks, and cylinders, a small set of key dimensions is sufficient. If each dimension is an independent variable, then we can produce a particular solid within a class by specifying a few key dimensions or parameters. In this modelling theory, many manufactured parts can be grouped into a class of families of similar shapes, where individual members of a family are distinguished by a few parameters. A single family of shapes is a generic primitive, and individual members are called instances or parameterised primitives. Formal definitions of some basic parameterised primitives are shown below,

$\vdash$ BLOCK $(dx, dy, dz)(l, w, h) =$

$\{p \mid \exists e.$ PRIMITIVE $e \wedge p \in e \wedge$

(FAR_SPACE (b e)$(1, 0, 0, dx)$) $\wedge$ (NEAR_SPACE (b e)$(1, 0, 0, l + dx)$) $\wedge$

(FAR_SPACE (b e)$(0, 1, 0, dy)$) $\wedge$ (NEAR_SPACE (b e)$(0, 1, 0, w + dy)$) $\wedge$

(FAR_SPACE (b e)$(0, 0, 1, dz)$) $\wedge$ (NEAR_SPACE (b e)$(0, 0, 1, h + dz)$)$\}$

$\vdash$ CYLINDER_Z $(dx, dy, dz)(r, h) =$

$\{p \mid \exists c.$ PRIMITIVE $c \wedge p \in c \wedge$

(NEAR_SPACE (b c)$(0, 0, 1, dz)$) $\wedge$ (FAR_SPACE (b c)$(0, 0, 1, h + dz)$) $\wedge$

(SQR(X(p) $+ dx$) + SQR(Y(p) $+ dy$) $\leq$ SQR r)$\}$

Translational movements of pritimive solids are modelled by the increments dx, dy, dz to represent displacements along the x, y, and z axis respectively. The CYLINDER_Z defines a cylindrical object placed along the z-axis. Primitives along other axes are similarly specified by its diameter, height and appropriate translational movements. Definitions of other solid primitives are given in Annex D.

7.7 Combinations of Solid Primitives

It was noted in section 7.1 that we wish to model interfaces and volumetric operations on solids, as well as the results of such operations on solids. It is obvious that if we model solids by closed regular sets [13] then we must model operations on solids by means of mathematical operations which take regular sets into closed regular sets. In other words, all set operations have to preserve the dimensionality of the solids which undergo these operations. This implies that the usual set operators may not be used because they destroy regularity. A set s is REGULAR iff it is equal to the CLOSURE of its interior subset, that is

$$\vdash \forall s.\ \mathsf{REGULAR}\ s = \mathsf{CLOSURE}\ (\mathsf{INTERIOR}\ s)$$

where the CLOSURE and INTERIOR functions are defined in section 7.5.4 and 7.5.5 respectively. We can consider intuitively that a closed regular set is an open set (its interior) covered with a tight skin, each point in the interior is therefore completely surrounded by other points in the set.

In other words, all geometric objects under consideration are those being defined as closed sets of points having a boundary subset and a finite interior subset. Boolean operations, such as set intersection, union and difference, are used to combine simple objects to form more complex ones. The algorithms that perform these operations must produce objects that are also closed sets of points and preserve the dimensionality of the initial objects. The latter requirement means that in any Boolean operation, such as a INTER $b = c$, all the objects must be of the same spatial dimension.

In fact, it is possible that the set operations on well-defined objects can produce a degenerate result which is not a solid object. The following section illustrates one such example.

[13] Regularity means three-dimensionally homogenous [97].

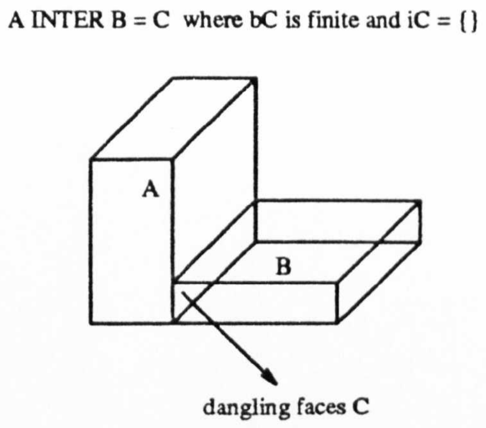


Figure 7.8: Dangling faces

7.7.1 Modified boolean operations

Let us consider the set operations on well-defined objects shown in Fig.7.8. A and B are well-defined because each possesses a boundary set b A and b B and an interior i A and i B. The resulting intersection is mathematically correct but geometrically improper, because C has no interior. Thus C is not like A and B, it is not a solid object, and the intersection operation does not preserve dimensionality. It follows that the concept of modified boolean operators are required to recognise this condition and produce an EMPTY set.

Let us consider the example shown in Fig.7.8 in more detail. Suppose that A and B are both well-defined objects, that is closed and dimensionally homogenous. Thus, A and B can be expressed as

$\vdash A = \mathsf{b}\ A\ \mathsf{UNION}\ \mathsf{i}\ A$

$\vdash B = \mathsf{b}\ B\ \mathsf{UNION}\ \mathsf{i}\ B$

Next, translate A and B into position prior to combining them by Boolean operation to form object C. First, perform the set-theoretic intersection, with the result shown in Fig.7.8. It should be noted that the dangling edge is obviously not dimensionally homogenous, yet it is a correct set intersection. To produce the expected result, which is obviously an EMPTY set in this case, the set intersection operation is modified such that it always gives a resultant set which is regular. The modified set intersection denoted by REG_INTER is defined in HOL as follows,

$\vdash \forall s\ t.\ s$ REG_INTER t = REGULAR (s INTER t)

To illustrate how the modified operation can achieve regularity, let us consider another example which is shown in Fig.7.9. From set intersection we have,

C = A INTER B

Rewriting this with the closed-set definitions, we have

C = (b A UNION i A) INTER (b B UNION i B)

which can be expanded to,

C = (b A INTER b B) UNION (i A INTER b B) UNION
  (b A INTER i B) UNION (i A INTER i B)

The geometric interpretation of each of the above four terms is illustrated in Fig.7.9. Since C = b C UNION i C we have to find the subsets of b C and i C that form a closed, dimensionally homogenous object C. We can easily recognise the dimensional interior of C from Fig.7.9d that is,

$\vdash$ i C = i A INTER i B

The problem now is that we have to determine b C, where b C is a valid combination of b A and b B. Notice that the boundaries of any new object will always consist of boundary segments of the combining elements. We can generalise this observation as follows: boundary points can become interior points, whereas interior points cannot become boundary points. Further more, for regularised

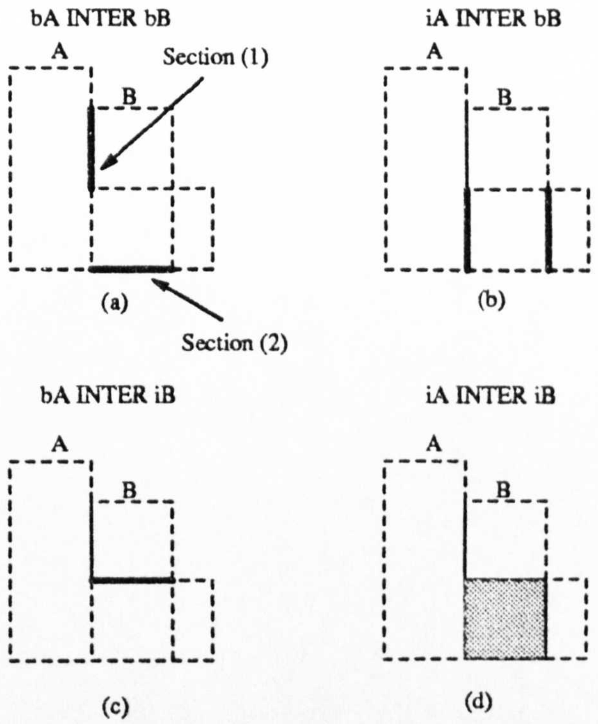


Figure 7.9: Geometric components of A REG_INTER B

intersections, we have proved that [14],

$\vdash$ (i A INTER b B) $\subset$ b C

$\vdash$ (b A INTER i B) $\subset$ b C

So far, we have accounted for Fig.7.9*b*-Fig.7.9*d* of the set combination for the regularised intersection. The problem now is to analyse (b A INTER b B) shown in Fig.7.9*a* to determine which of its subsets are a valid set of the boundary of C. Consider two identical overlapping intersections of A and B. These two sections both belong to (b A INTER b B), however it can be shown that, section (2) only includes points that belong to both the INTERIOR of A and the CLOSURE of B, and it therefore can be identified by the relation,

(b A INTER b B) INTER CLOSURE (i A INTER CLOSURE B)

[14]Proofs are shown in Annex D

In fact, the following theorem defining the boundary subset of regularised intersection has been proved in Annex D,

```
⊢ ∀s t. BOUNDARY (s REG_INTER t) =
      (BOUNDARY s INTER INTERIOR t) UNION
      (BOUNDARY t INTER INTERIOR s) UNION
      (BOUNDARY s INTER BOUNDARY t INTER
          CLOSURE (INTERIOR s INTER CLOSURE t))
```

By following the same arguments and proof procedures as shown above, definitions for regularised union, difference and complement of regular sets can be shown as follows,

```
⊢ ∀s t. s REG_UNION t = REGULAR (s UNION t)
⊢ ∀s t. s REG_DIFF t = REGULAR (s DIFF t)
⊢ ∀s. REG_COMP s = REGULAR (COMP s)
```

and consequently, theorems to evaluate their resultant BOUNDARY have been proved as follows [15],

```
⊢ ∀s t. BOUNDARY (s REG_UNION t) =
      (BOUNDARY s INTER COMP t) UNION
      (COMP t INTER BOUNDARY t) UNION
      (BOUNDARY s INTER BOUNDARY t INTER
          CLOSURE (COMP s INTER COMP t))
⊢ ∀s t. BOUNDARY (s REG_DIFF t) =
      (CLOSURE s INTER BOUNDARY (COMP t)) UNION
      (INTERIOR s INTER BOUNDARY t) UNION
```

[15]These theorems are produced by geometrical considerations and then formally verified in HOL as shown in Annex D.

$$\text{(BOUNDARY } s \text{ INTER BOUNDARY } t \text{ INTER}$$
$$\text{CLOSURE(INTERIOR } s \text{ INTER COMP } t\text{))}$$
$$\vdash \forall s\, t.\, \text{BOUNDARY (REG\_COMP } s) = \text{BOUNDARY } s$$

So far, it has been formally shown that by representing physical objects in a closed point-set topology and applying modified boolean operations to these subsets, we can ensure that the homogenous dimensionality of the combined object is preserved. Let us now turn our attention to the problem of specifying complex solid objects using these boolean operations. The following section discusses a formal scheme for combining solid primitives based on a representation called a Boolean model.

7.7.2 Boolean model

A representation is a Boolean model when a solid object is represented by the modified boolean combination of two or more simpler objects. If A, B and C denote solids and if $C = A \langle \text{OP} \rangle B$, where $\langle \text{OP} \rangle$ is any modified boolean operator, then $A \langle \text{OP} \rangle B$ is called a boolean model of C. It is clear from section 7.7 that as a result of the modified boolean operations, A, B and C are of the same spatial dimension.

Boolean models are, in fact, a generalisation of cell decomposition [18]. In cell decomposition models, individual cells are combined using a gluing operation, which is a limited form of the union operator where components are joined at only perfectly matched faces. Modified boolean operators are more versatile, since boundaries of joined components need not match and interiors need not be disjoint. Furthermore, in this model, all the modified boolean operators union, difference, and intersection are used, so that material can be added as well as removed.

Boolean models of objects are represented as ordered binary trees btree whose leaves or terminal nodes are either primitives or transformational motions. Each subtree of a node [16] represent a solid resulting from the combination and transformation operations indicated below it. The root represents the final object. Boolean trees may be defined by the following recursive definition,

$$\begin{aligned}\langle \text{btree} \rangle \;:=\; & \langle \text{primitive leaf} \rangle \;| \\ & \langle \text{btree} \rangle\, \langle \text{set operator node} \rangle\, \langle \text{btree} \rangle \;| \\ & \langle \text{btree} \rangle\, \langle \text{motion node} \rangle\end{aligned}$$

The semantic of btree representation is clear; each subtree that is not a transformation leaf represents a set resulting from the application of the indicated motional/combinational operators to the sets represented by the primitive leaves.

When the primitive solids of a boolean model are bounded, and hence are closed-sets, the algebraic properties of closed-sets guarantee that any btree is a valid representation of closed-sets if the primitive leaves are valid. Because primitive leaf validity may be easily assessed, the validity of a btree based on bounded primitives may be ensured essentially at the syntactical level.

It should be noted that the basic idea behind the boolean model of btree is not new. Boolean representations are natural and most suitable for representing and manipulating binary relations. In fact, a similar definition of btree has been used to define data type abstraction, for example, by Melham [79].

7.8 Solid Modelling in Transformer Designs

Since there is no ambiguity in using a Boolean model to represent a 3-D shape, the importance of using solid modelling in mechanical designs is well recognised. This section discusses the use of such formal models in the design of mechanical parts of transformers. It should be noted that in designing mechanical parts using the formal model of btree as described in section 7.7.2 the design verification is implicit. This means that successive operations on solids will result in a valid object by using correctness-preserving transformations of the regularised boolean

[16] which is not a transformation leaf

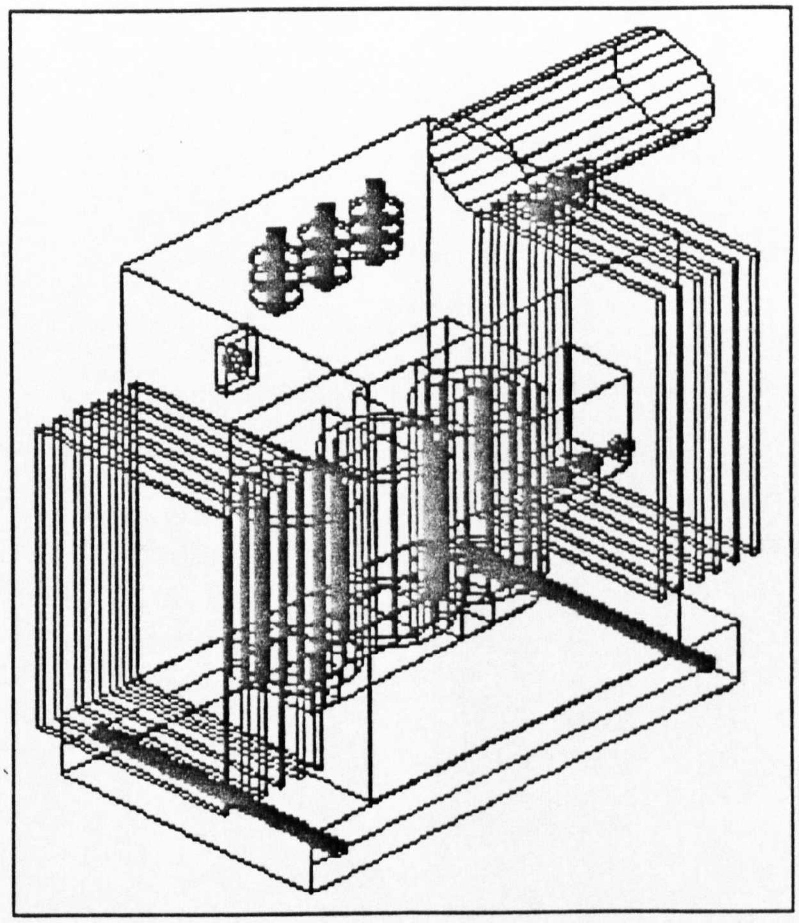


Figure 7.10: A geometric model of a power transformer

operations. In other words, by starting with valid objects such as pre-defined primitives and applying operations such as REG_INTER, REG_UNION, REG_DIFF and REG_COMP successively, it will always ensure a valid resultant object without the need to verify the rigidity of the final model. Let us consider the schema of the simplified transformer shown earlier in Fig.7.2. Its description can be formally specified in HOL as follows,

```
⊢ transformer_imp =
    let core = BLOCK (1375, 1000, 750)(1757, 750, 2000) REG_UNION
            BLOCK (1500, 1250, 500)(250, 250, 500) REG_UNION
            BLOCK (2750, 1250, 500)(250, 250, 500)) REG_DIFF
            BLOCK (1625, 500, 1000)(500, 2000, 1500)) REG_DIFF
            BLOCK (2375, 500, 1000), (500, 2000, 1500)) in
    let coil = (CYLINDER_Z (1500, 1375, 900)(1500, 300) REG_DIFF
            CYLINDER_Z (1500, 1375, 900)(1400, 400)) REG_UNION
            (CYLINDER_Z (2250, 1375, 900)(1500, 300) REG_DIFF
            CYLINDER_Z (2250, 1375, 900)(1400, 400)) REG_UNION
            (CYLINDER_Z (3000, 1375, 900)(1500, 300) REG_DIFF
            CYLINDER_Z (3000, 1375, 900)(1400, 400)) in
    let lugs = (BLOCK (750, 1500, 3600)(250, 100, 400) REG_DIFF
            CYLINDER_Y (875, 1400, 3800)(600, 75)) REG_UNION
            (BLOCK (3500, 1500, 3600)(250, 100, 400) REG_DIFF
            CYLINDER_Y (3625, 1400, 3800)(600, 75)) in
    let tank = BLOCK (1000, 500, 500)(2500, 2000, 3500) in
    let cooling = CYLINDER_X (0, 1400, 4250)(1500, 400) REG_UNION
            CYLINDER_Z (1250, 1700, 3900)(300, 100) in
    let cable = BLOCK (1875, 2500, 2750)(750, 500, 750) REG_UNION
            BLOCK (2000, 2625, 2500)(500, 250, 500) REG_UNION
            CYLINDER_Y (2000, 2750, 3000)(350, 50) REG_UNION
            CYLINDER_Y (2250, 2750, 3000)(350, 50) REG_UNION
```

```
          CYLINDER_Y (2500, 2750, 3000)(350, 50) in
    let bushing = CYLINDER_Z (3250, 1500, 4100)(50, 150) REG_UNION
          CYLINDER_Z (3250, 1500, 4250)(50, 150) REG_UNION
          CYLINDER_Z (3250, 1500, 4400)(50, 150) REG_UNION
          CYLINDER_Z (3250, 1500, 4000)(650, 50) REG_UNION
          CYLINDER_Z (2850, 1500, 4100)(50, 150) REG_UNION
          CYLINDER_Z (2850, 1500, 4250)(50, 150) REG_UNION
          CYLINDER_Z (2850, 1500, 4400)(50, 150) REG_UNION
          CYLINDER_Z (2850, 1500, 4000)(650, 50) REG_UNION
          CYLINDER_Z (2450, 1500, 4100)(50, 150) REG_UNION
          CYLINDER_Z (2450, 1500, 4250)(50, 150) REG_UNION
          CYLINDER_Z (2450, 1500, 4400)(50, 150) REG_UNION
          CYLINDER_Z (2450, 1500, 4000)(650, 50) REG_UNION in
    let radiator = BLOCK (850, 750, 750)(50, 1500, 2500) REG_UNION
          BLOCK (700, 750, 750)(50, 1500, 2500) REG_UNION
          BLOCK (550, 750, 750)(50, 1500, 2500) REG_UNION
          BLOCK (400, 750, 750)(50, 1500, 2500) REG_UNION
          BLOCK (250, 750, 750)(50, 1500, 2500) REG_UNION
          BLOCK (100, 750, 750)(50, 1500, 2500) REG_UNION
          BLOCK (3650, 750, 750)(50, 1500, 2500) REG_UNION
          BLOCK (3800, 750, 750)(50, 1500, 2500) REG_UNION
          BLOCK (3950, 750, 750)(50, 1500, 2500) REG_UNION
          BLOCK (4100, 750, 750)(50, 1500, 2500) REG_UNION
          BLOCK (4250, 750, 750)(50, 1500, 2500) REG_UNION
          BLOCK (4400, 750, 750)(50, 1500, 2500) REG_UNION in
    let base = (BLOCK (500, 500, 0), 3500, 2000, 500) REG_DIFF
          CYLINDER_Y (750, 0, 250)(4000, 50)) REG_DIFF
          CYLINDER_Y (3750, 0, 250)(4000, 50) in
BUILD [core, coil, lugs, tank, cooling,
```

$cable, bushing, radiator, base]$ REG_UNION

where BUILD is a higher-order recursive function of the form,

$\vdash$ BUILD [] = $\lambda f.\ \{\}\ \wedge$

BUILD (CONS $hd\ tl$) = $\lambda f. hd\ f$ BUILD $[tl]$

The function BUILD takes a list of closed point-sets as its argument and applies the infix function f recursively to its components. In this case, f represents the regularised operators. If the argument list of BUILD is an empty list, it is obvious that the function will return a null set { }.

The diagram shown in Fig.7.10 is a bit-map image of a power transformer specified by the above formal specification. The solid modeller is implemented in Pascal and it operates in batch processing mode. A detailed description of the system and the actual inputs for the transformer model in Fig.7.10 is included in the Annex A. The size of the specification in this example has shown that, to achieve the desired integrity and validity of the intended model, the formal descriptions of mechanical components can become very complex, even for a simplified description of devices. The verification therefore becomes complicated, and the solid modeller is built in this case to automatically keep track and verify the design steps.

However, the above specification is in fact incomplete, since in the design of transformers, as well as in other solid modelling applications, there is a generic problem which requires further considerations, namely null object detection.

7.8.1 Null-object detection

The null-object detection problem can be stated as follows: given a representation R of an object s determine from computations on R whether or not $s = \{\ \}$. As a matter of fact, null-object detection is important mainly for interference detection, which can be stated as follows: given two solids A and B, is their common volume

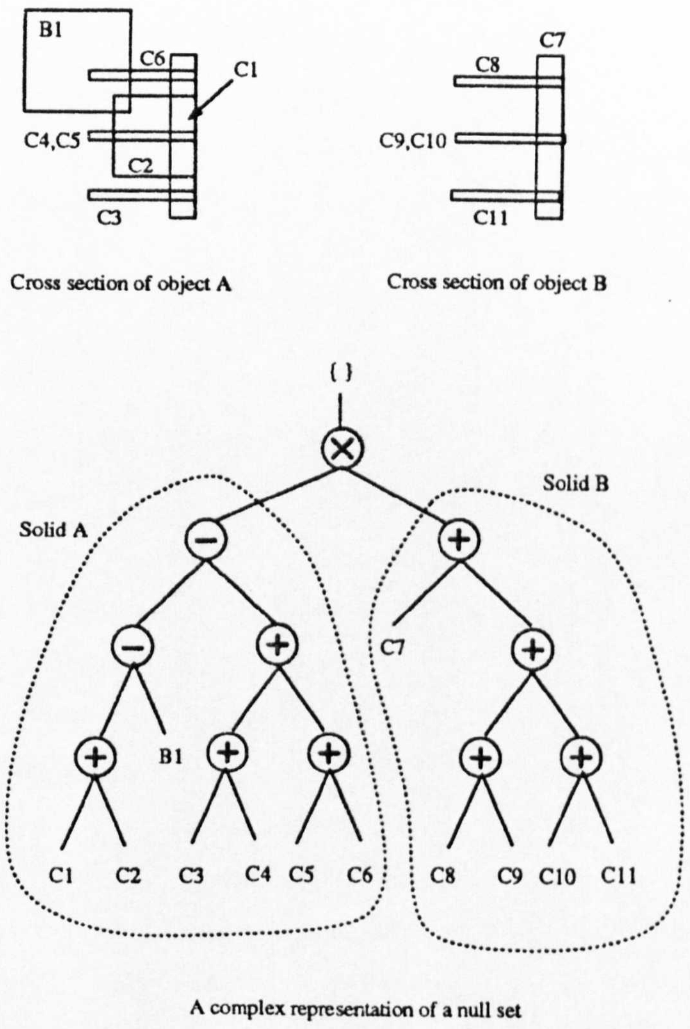

B1
C6
C1
C4,C5
C2
C3
Cross section of object A
C7
C8
C9,C10
C11
Cross section of object B
{ }
Solid A
Solid B
C7
B1
C1 C2 C3 C4 C5 C6
C8 C9 C10 C11
A complex representation of a null set

Figure 7.11: An example of a null object

empty ? [17]. The detection of null-objects can be specified using the function BUILD as follows,

$$\vdash \forall\, s.\ \mathsf{NULL\_DETECT}\ s = \mathsf{BUILD}\ s\ \mathsf{REG\_INTER} = \{\,\}$$

where s represents a list of valid primitives. In the above example, we have described the mechanical components of the transformer without making any assertions about its relative spatial properties, for instance, we need to ensure that the components of the device should not geometrically interfere with each other, i.e. they do not have a common volume, that is

$$\vdash \mathsf{BUILD}\ [core, coil, lugs, tank, cooling,$$
$$cable, bushing, radiator, base]\ \mathsf{REG\_INTER} = \{\,\}$$

The set operators provide a powerful means of defining objects and also for modelling some important phenomena - intersection for spatial collision or common volume, as noted earlier, and set difference for material removal, as in machining. Unfortunately, representation of btree does not always give trivial null-object detection algorithms because the null set can have arbitrarily complex btree representations as shown in Fig.7.11 [18]. The tree in Fig.7.11*b* is a typical btree representation of the null set. It corresponds to the volume common to a pair of solids that fit together as shown in Fig.7.11*a*.

Null-object detection is therefore a non-trivial problem because the null set (which is geometrically very simple) can have arbitrarily complex btree structures. The approach presented here seeks to reduce (or simplify) the btree of the null set by removing redundant primitives. The following section describes a formal mechanism from which manufacturing process can be manipulated without loss of the integrity of the solid objects.

[17] The common volume is in fact another solid in mathematical terms

[18] To avoid cluttering the diagram, the symbol $+$, $-$, and $\times$ are used to denote REG_UNION, REG_DIFF and REG_INTER respectively.

7.8.2 Specifying the manufacturing process

One of the most important applications of solid modelling theory, is the simplification of manufacturing processes. Solid modelling, as has been shown in earlier sections, is suitable as a mechanism for describing manufacturing procedures, since Boolean operations such as intersection, union and difference between solid primitives are analogous to industrial processes of welding, cutting and drilling operations. Therefore, most of manufactured assemblies can be easily described using the boolean model semantic, as described in section 7.7.2, however, the modelling theory can be used to optimise the manufacturing process. For instance, consider the procedure of manufacturing the object shown in Fig.7.12*a*, in which the object is represented by the corresponding btree. The modelling theory can improve the manufacturing process by rearranging the structure of btree and grouping together those operations of the same type or of practical preference. Such rearrangement or simplification can be formulated in the following two steps,

1. Tree reconstruction in which all participating nodes of the feature are located and grouped together to form a subtree.
2. Tree transformation in which the subtree resulting from step 1 is replaced by an equivalent subtree.

Both steps are subject to the condition that the object must remain the same. Their difference is that each node is relocated in step 1 but is replaced by a different new node in step 2. To implement the node relocation algorithm, we replace each algebraic representation of the boolean function associated with btree by its positive form [19], which uses only regularised intersection and union operators but represents rigorously the same boolean function and thus the same solid as the original tree. The regularised operators, union, intersection and complement, form a boolean algebra over regular sets. Therefore, De Morgan's law can be invoked to manipulate btree expressions.

[19] A positive form of a btree is one which does not include REG_DIFF operations.

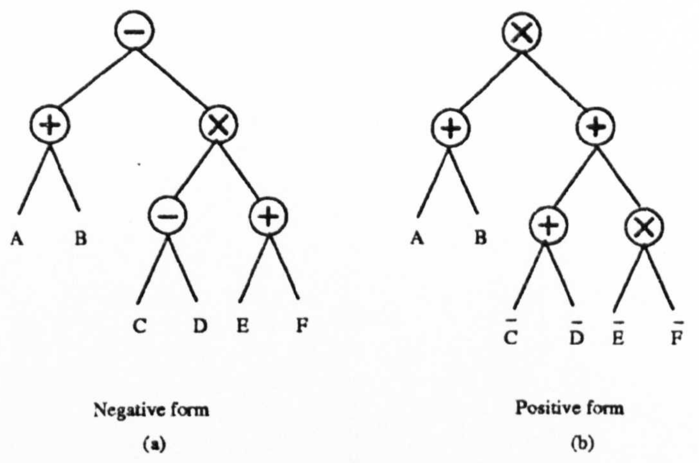


Figure 7.12: An example of process simplification

A btree can be converted into its positive form by a preorder traversal of the tree applying, where appropriate, the following transformations to each node [20],

$\vdash \forall xy.\, x$ REG_DIFF $y = x$ REG_INTER (REG_COMP y)

$\vdash \forall xy.$ REG_COMP (x REG_UNION y) =

(REG_COMP x) REG_INTER (REG_COMP y)

$\vdash \forall xy.$ REG_COMP (x REG_INTER y) =

(REG_COMP x) REG_UNION (REG_COMP y)

$\vdash \forall.$ REG_COMP (REG_COMP x) = x

This reformulation, which is illustrated in Fig.7.12, exchanges the REG_INTER and REG_UNION operators at negative internal nodes, replaces REG_DIFF operator by REG_INTER if the corresponding node is positive and by REG_UNION otherwise, and replaces negative primitives by their regularised components. The resulting positive form contains only REG_INTER and REG_UNION operators. In a positive form, all nodes are positive, but may represent unbounded regular sets. However, the set represented by the entire positive form is equal to the solid represented by the original btree, and is hence bounded. The btree in Fig.7.12 is verified by the

[20]The proof of these theorems are shown in Annex D

following theorem, its proof is shown in Annex D,

$$\begin{aligned}\vdash\ & (a\ \mathsf{REG\_UNION}\ b)\ \mathsf{REG\_DIFF} \\ & \quad ((c\ \mathsf{REG\_DIFF}\ d)\ \mathsf{REG\_INTER}\ (e\ \mathsf{REG\_UNION}\ f)) = \\ & \qquad (a\ \mathsf{REG\_UNION}\ b)\ \mathsf{REG\_INTER} \\ & \qquad (((\mathsf{REG\_COMP}\ c)\ \mathsf{REG\_UNION}\ d)\ \mathsf{REG\_UNION} \\ & \qquad ((\mathsf{REG\_COMP}\ e)\ \mathsf{REG\_INTER}\ (\mathsf{REG\_COMP}\ f))\end{aligned}$$

7.8.3 Computation of overall volume

The volume of a designed solid is conventionally calculated using a mechanical drawing and 3-D models. Using information derived from solid models, the overall volume of the resulting transformer can be verified against its original specification. The overall dimensions of the compound object can be inferred from the overall dimensions of its components as specified below,

$$\begin{aligned}\vdash\ & \mathsf{OVERALL\_B}\ (dx, dy, dz)(l, w, h)) = \\ & \quad \{p \mid \mathsf{X}(p)\ \mathsf{within}\ (dx, l + dx)\ \land \\ & \qquad \mathsf{Y}(p)\ \mathsf{within}\ (dy, w + dy)\ \land \\ & \qquad \mathsf{Z}(p)\ \mathsf{within}\ (dz, h + dz)\}\end{aligned} \tag{7.17}$$

where within is an infix function which asserts whether the corresponding coordinate is included in the specified interval,

$$\vdash \forall n.\, n\ \mathsf{within}\ (k, p) = k < p\ \land\ k \leq n\ \land\ n \leq p$$

Overall dimensions of cylindrical parts can be also be deduced from their rectangular bounding box as follows,

$$\begin{aligned}\vdash\ & \mathsf{OVERALL\_CX}\ (dx, dy, dz)(r, h) = \\ & \quad \mathsf{OVERALL\_B}\ (dx, dy, dz)(h, 2 * r, 2 * r)\end{aligned} \tag{7.18}$$

$$\vdash \mathsf{OVERALL\_CY}\ (dx, dy, dz)(r, h) =$$

$$\text{OVERALL\_B}\ (dx, dy, dz)(2*r, h, 2*r) \tag{7.19}$$

$$\vdash \text{OVERALL\_CZ}\ (dx, dy, dz)(r, h) =$$
$$\text{OVERALL\_B}\ (dx, dy, dz)(2*r, 2*r, h)) \tag{7.20}$$

The overall volume of the resulting object is derived by replacing the solid primitives by their overall counterparts. Inference rules for manipulating the ranges of coordinates can easily be derived in HOL using the well-order properties of integers as follows [21],

$$\vdash \forall n\ a\ b\ c\ d.(n\ \text{within}\ (a,b)\ \wedge\ n\ \text{within}\ (c,d)) \supset$$
$$(a \le c\ \wedge\ d \le b)\ \Rightarrow\ n\ \text{within}\ (a,b)\ |\ n\ \text{within}\ (a,d) \tag{7.21}$$

$$\vdash \forall n\ a\ b\ c\ d.(n\ \text{within}\ (a,b)\ \vee\ n\ \text{within}\ (c,d)) \supset$$
$$(a \le c\ \wedge\ d \le b)\ \Rightarrow\ n\ \text{within}\ (a,b)\ |\ n\ \text{within}\ (a,d) \tag{7.22}$$

$$\vdash \forall n\ a\ b\ c\ d.(n\ \text{within}\ (a,b)\ \wedge\ \neg(n\ \text{within}\ (c,d))) \supset$$
$$(c \le b\ \wedge\ d < b)\ \Rightarrow\ (n\ \text{within}\ (a,c)\ \wedge\ n\ \text{within}\ (d,b)\ |$$
$$(c \le b\ \wedge\ b \le d)\ \Rightarrow\ (n\ \text{within}\ (a,c)\ |\ n\ \text{within}\ (a,b)) \tag{7.23}$$

It should be noted that theorems (7.21-7.23) are used to derive the overall dimensions of objects combined by regularised operators REG_UNION, REG_INTER and REG_DIFF respectively.

Fig.7.13 illustrates the overall volume of a simple compound object. Formally, the volume is specified by the relation,

$$\vdash \text{volume} = \text{OVERALL\_B}\ (\text{BLOCK}\ (1,0,1)(4,4,4))\ \text{REG\_UNION}$$
$$\text{OVERALL\_CY}\ (\text{CYLINDER\_Y}\ (4,0,4)(2,3)) \tag{7.24}$$

By rewriting (7.24) with (7.17-7.23) consecutively, we have,

[21] Proofs of these theorems are included in Annex D

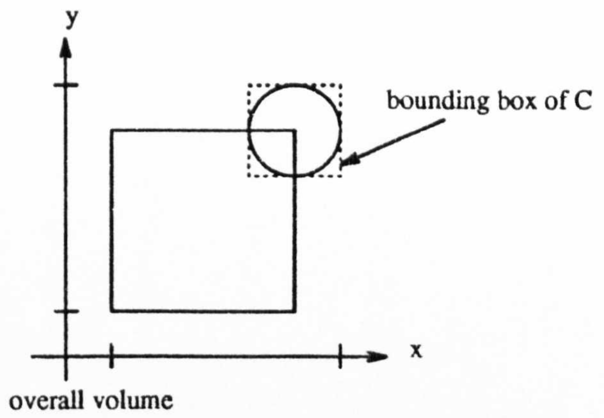


Figure 7.13: An example of volume verification

$$\begin{aligned} \vdash \mathsf{volume} = \{p' \mid\ & 1 \leq \mathsf{X}(p)\ \wedge\ \mathsf{X}(p) \leq 6\ \wedge \\ & 0 \leq \mathsf{Y}(p)\ \wedge\ \mathsf{Y}(p) \leq 4\ \wedge \\ & 1 \leq \mathsf{Z}(p)\ \wedge\ \mathsf{Z}(p) \leq 6\} \end{aligned}$$

This means that the space occupied by the resulting object is defined by a closed point-set which satisfies the volume relation. In fact, the algorithm does not give an explicit numerical volumetric result for the occupied space, however the dimensions indicated by the relation volume can be used for verification against the original specification. In practice, the derivation of the overall volume can be automated using an ML function which maps the solid components to their corresponding *bounding* boxes and subsequently rewrites the resultant set with theorems (7.17-7.23).

7.9 Discussion

In this chapter, a mechanisation of a solid modelling theory in higher-order logic was undertaken with the aim of facilitating reasoning about solid properties. The mechanisation was entirely carried out using the HOL theorem prover so all the theorems shown in this work were proved mechanically from their definitions using the HOL system.

This work focuses on the representational issues that arise in the design and analysis of solid models. The distinguishing feature of the present treatment is the clear distinction between a physical solid and its abstract mathematical model. The distinction has several advantages. Specifically,

- Mathematical models can be studied and reasoned rigorously and independently of computational considerations, and
- Important concepts such as model validity and ambiguity can be defined mathematically with the assistance of a mechanical theorem prover.

A fundamental requirement for the effective use of model abstraction in reasoning about its behaviour is, of course, a formal representation of the model itself. This chapter has shown how solid objects can be characterised formally in higher-order logic, and has given examples to illustrate how this model can be used to support reasoning about physical properties.

A common theme of these examples is the importance of an appropriate choice of abstraction for the polymorphic type of sets. For instance, in section 7.6, a simple tuple of the type *point* was used to formulate a boolean model for combining objects.

The formal specification of the solid model addresses several aspects which hitherto were only informally stated in user manuals for graphic systems. The model for specifying the object is novel. The construction of the invariant btree schema brings out the fact that even at an abstract level, solid modelling systems are inherently complex to specify.

There are a number of related work on formal specification of graphics systems. The work reported by Mallgren [76] is most closely related to the particular formulation of BUILD transformation described in section 7.8.1. Mallgren defines four general concepts which he uses as the basis for the specification of graphics data types, region, picture, graphic transformation and hierarchical picture structure. Each concept is treated as a collection of objects operated upon by well-defined operations. For example, the key operation on a hierarchical picture structure is display, which converts it into the resulting picture.

The main difference between these two approaches is the fact that all the geometric abstractions defined in this chapter are concerned with 3-D representations of solids while the basis of Mallgren's work is specialised for concepts to incorporate into 2-D graphical display of objects. The argument against his approach is that the modeller has to work with abstractions that do not correspond to those normally used in drawing and in consequence they have to be aware of representations at an inappropriate level of detail. Also responsibility for the integrity of the structures constructed rests with the modeller. The approach used in this work however, relies on the sequence of correctness-preserving transformations to maintain the integrity and validity of the resulting solids, and consequently the verification of the model is implicit.

Finally, we have shown that formal specification help us to reason about the modelling of rigid solid objects, both formally and informally, but the work is incomplete. For example, we have dealt with only partial correctness.

An area we have not touched upon is the specification of the modelling system at the user dialogue level. Such a specification represents the highest level abstraction, in that only the behaviour of the interface as it is seen by the user is determined.

Chapter 8

Concluding Remarks

In this chapter we evaluate the research described in this thesis. We present a summary of the results achieved, discuss the problems arising from the research and suggest solutions to these problems as well as ideas for future research.

8.1 Summary of Thesis

The goal of the research described in this thesis was to investigate the extent to which a subset of formal methods, namely Higher Order Logic could be used to specify safety-related systems. It enables engineers to design systems within a formal verification environment, with the aim of obtaining correct specifications earlier in the design process. The research was undertaken using the HOL theorem proving system and can be summarised as follows.

8.1.1 Methodology

A methodology for using formal techniques of specification and verification in the design process has been presented. A crucial concern is how the target system acts on the plant and, in particular, how a failure of the system can bring the plant into a state where an accident is likely to occur. Not all system failures lead to accidents. The progression from a system failure to destruction goes via an intermediate, dangerous state of the plant. This state may lead to an accident, but in principle, the plant may be returned to a safe state. Indeed, this restoration is the goal of the safety system.

It is important to realise that hazards may continue to exist despite the safety system. Each hazard has a certain risk associated with it. The purpose of a safety system is to reduce the risk associated with the hazards to a value below an acceptable level, according to some appropriate criteria. By definition, a system will be safe if the level of risk is less than the agreed acceptable value. In fact, there are five main principles in the design methodology described in this thesis. They can be summarised as follows,

- **Independent analysis of system safety:** In order to establish safety requirements, one has to identify the possible impact the plant has on its environment. This hazard analysis step aims to identify and evaluate hazards and also to identify the general safety criteria to be used in the design.
- **Structuring according to criticality:** The separation of the system functions into safety classes is used to guide the design which should aim to preserve this separation at the modular level. Having preserved this separation, extra effort can be devoted to increase the quality of the safety-critical modules without the risks of incurring excessive development costs.
- **Achieving high reliability of the safety-critical modules:** The previous principle states that the system structure should support the division between safety-critical and non safety-critical functions. If such a separation is guaranteed then the primary concern is the correctness and reliability of the safety critical parts. This concentration on the safety-critical parts of the target system offers two benefits. The first is that the scale of the problem is reduced, since the safety critical part is usually only a small proportion of the complete system. The second is that since timescales and budgets are usually limited, it delineates those parts of the system which need special attention from the rest. However, this is not to say that the quality of the rest of the system is of no importance, but it has to be recognised that it is neither possible nor cost effective to build the whole system to the same

quality level.

- **Design for safe states:** To achieve system safety, the possibility of bringing the system into a safe state, or into a state of reduced risk, should be investigated. Such safe states are states where other quality factors (eg. reliability, risk, ...) are considerably reduced.

- **Continuous monitoring of safety:** Despite the effort to increase reliability of the safety critical components, the system itself and the environment should be continuously monitored to intercept failures before they evolve into accidents. This principle is based on the recognition of the fact that no presently available means can guarantee absolute safety.

8.1.2 Specification techniques

A formal specification technique has been presented to show that the formal design rules, in the form of heuristics, can be efficiently formalised in Higher Order Logic.

First, an informal description of this design style was given to highlight various aspects which need to be formalised. Then a complete list of heuristic rules for designing practical systems was compiled. Engineering systems are intrinsically classified into two subsets, namely memory bearing systems and memoryless systems. Based on this classification, a method was presented to derive the correctness statements for classes in the system. This method relies on the fact that the form of the correctness statement developed is uniform across the whole range of engineering problems.

The use of these formal techniques were then demonstrated through a number of worked examples, ranging from a simple model of a timing marker to the complete specification of a CAD package. In each case the form of the derived correctness statement was identical. This is an important factor in using such formal design techniques. If the statement of correctness has the same general form at each level, then the arbitrary mixing of complex and trivial problems can be done with confidence.

8.1.3 Timing analysis

The process of generating and verifying the timing conditions of a system is called timing analysis. Timing analysis is important at all levels of description. In performing temporal abstraction we generate a number of timing conditions which must hold if the system is to achieve the intended performance criteria. Having formally defined temporal functions, and shown that they construct a well defined time mapping for any predicate that is true at an infinite number of points of concrete time, it is possible to formulate a correctness theorem based on temporal abstraction by time mapping. For instance, if p is an appropriate predicate that indicates which points of concrete time correspond to points of abstract time, then a correctness theorem that relates a design model $M[sig_1, ..., sig_n]$ to an abstract specification $S[s_1, ..., s_n]$ can be formulated as follows,

$$\vdash M[sig_1, ..., sig_n] \supset S[sig_1 \text{ When } p, ..., sig_n \text{ When } p]$$

This correctness theorem states that whenever the signals $sig_1, ..., sig_n$ satisfy the model, the abstract states $s_1, ..., s_n$ constructed by projecting on $sig_1, ..., sig_n$ when the predicate p is true will satisfy the temporally abstract specification. In general, the predicate p can be defined in terms of the variables $sig_1, ..., sig_n$ to make the times at which the values in the model are projected depend on the behaviour of the device itself. This approach is illustrated in a case study described in Chapter 6 using a theory of temporal abstraction. The theory of temporal abstraction is defined in the general purpose typed lambda calculus used by the HOL theorem prover. The theory is not only applied using HOL, it is derived in it as well. This ensures the validity of the theory. Polymorphic typing in the logic allows the theory to be totally independent of the representation of the actual implementation.

8.1.4 Specifying behaviour

Two case studies were presented to illustrate how formal techniques can help in the specification and the design of systems. The first of these is an automotive device, DBW, which is useful in the design of memory bearing systems, and closed-loop control systems. Based on a simple top level specification of this device, a formal proof of correctness was carried out illustrating how such formal techniques can help to improve and sharpen the final specification in order to verify that the design meets its intended safety requirements and integrity level.

The last case study shows a new design for a CAD drawing package. This has been designed down to the primitive level using the design style described earlier on, and has been simulated using a simple drawing simulator. A fundamental requirement for the effective use of model abstraction in reasoning about its behaviour is, of course, a formal representation of the model itself. In this case study, it has been shown that solid objects can be characterised formally in higher-order logic, and we have presented examples to illustrate how this model can be used to support reasoning about physical properties.

The formal specification of the solid model addresses several aspects which hitherto were only informally stated in user manuals for graphic systems. The model for specifying the object is novel. The construction of the invariant btree schema brings out the fact that even at an abstract level, solid modelling systems are inherently complex to specify. Furthermore, in this case study, although we have shown that formal specifications aid in reasoning about the modelling of rigid solid objects, both formally and informally, the work is incomplete. For example, we have dealt with only partial correctness. An area we have not touched upon is the specification of the modelling system at the user dialogue level. Such a specification represents the highest level abstraction, in that only the behaviour of the interface as it is seen by the user is determined.

8.2 Discussion

In the following sections, some of the points above are discussed further, along with suggestions on how future research may proceed to develop and improve the ideas.

8.2.1 Correctness of a real design

In the case studies we have specified and verified a real system design. By real we mean that the DBW system was designed and fabricated for a practical purpose, independent of the case study. The DBW system was first manufactured in 1990 by Econocruise Limited. A commercial version of the system is now available [111].

Modelling and verifying a real design brings forwards issues not dealt with in contrived examples. It is dangerous, however, to claim that a design has real-world correctness. If the mathematical system is sound then a proof of correctness is valid. However, the mathematical system employs models of real devices and their interconnections. The user must accept that the proofs are based on these models and do not verify the behaviour of real devices outside these models. This is no different from other CAD tools, which are all based on models which the user can accept or reject as accurate models.

In the real world the DBW system sometimes malfunctions. The malfunctions are, for example, due to badly behaved signals from its environment. The partial specification in HOL demands certain behaviour from the environment to ensure the desired output behaviour, the behaviour in the presence of badly behaved environment signals is not described. The malfunctions of the DBW do not reflect on the validity of the formal proofs of correctness. In the presence of well behaved signals the system functions correctly. It is important however that the user appreciates the limitations as well as the power of the formal techniques.

8.2.2 Relating different levels of abstraction

A methodology for using formal techniques of specification and verification in the design process has been presented. We have indicated above some of the difficulties found in these case studies. Although many problems could be avoided by a restricted design style, we believe that some basic problems endure. It is difficult to devise an accurate but abstract formal specification of a complex device. It is necessary to refine the formal specification at the higher levels as the specifications at lower levels become fixed.

The end product of formal verification is a theorem stating that a behavioural specification is a logical consequence of a structural specification and assumptions about the behaviour of a small set of primitive components which can be implemented as physical devices. While formal verification offers greater certainty than simulation, the mere generation of such a theorem is not completely satisfactory evidence that a design is correct. First, validity of the theorem depends on the behavioural models used for primitive components and external devices. Second, the behavioural specification may be misread or its full implications may not be fully understood. Complete solutions for this second problem are more difficult to propose. However, the problem is partially solved by relating behaviour to increasingly abstract levels such as the semantics of a programming language.

8.2.3 Verification process

A sceptical view of the formal design lifecycle suggests that the automation of the verification process is impractical because the process simply cannot be completely automated. In our view, complete automation would undermine the real value of formal verification. As part of a system design methodology, the actual process of verifying a design may be of more value than the end product. Constructing a formal proof forces a designer to consider exactly why a design is correct. As suggested in [27], an untidy step in the proof may indicate the need for a design modification. Furthermore, the designer is more likely to understand the full implications of a behavioural specification after the effort of producing the proof. Powerful tools such as the HOL system automate most of the tedious steps in a large proof. This allows a designer to reason efficiently about a system specification. The underlying formalism ensures that this reasoning process is absolutely correct.

8.3 Future Work

In this section, some subtle aspects of the work in this thesis are discussed and further research in the related areas are proposed.

8.3.1 Executable specifications

A major obstacle in the development of these examples was the inability to execute the formal specification. Several errors in the design and specification of the case studies were discovered in the course of verifying their correctness, many of these errors would have been revealed earlier if the specifications were executable. Verification was an efficient way to find some of the more obvious errors in the initial specification. Camilleri [25] points out the advantage of executing formal specifications as a means of supporting a verification methodology. In this respect, the Boyer-Moore theorem prover [17] offers the advantage of directly executable specifications. The increased expressiveness of higher order logic results in terms which do not always have an obvious interpretation as an executable specification. However, the implementation of an executable subset of higher order logic is a foreseeable addition to the HOL system.

8.3.2 Computer aided design with formal methods

We have described formally how to build solid components and derive their functional behaviour from top-level descriptions. This only addresses one aspect of the design cycle. This approach should be incorporated into a computer aided design system which encourages designers to apply formal methods from design conception to final implementation. A proposal for such a system is described below. Subrahmanyam [106] describes an expert system for VLSI design which incorporates formal methods as well as more conventional tools. This is an evolutionary approach. The system relates different levels of description formally, but the designer has the feeling of using tools he/she is familiar with. Milne [84] describes a more revolutionary approach, which does not attempt to incorporate existing tools, where a system can be described at various different levels from specification to layout and the different levels can be formally related.

A computer aided design system should consist of an interactive workstation which includes graphical tools as well as formal methods for reasoning about drawings. Such a workstation should provide tools for top-down and bottom-up design. In addition, such a system should keep the various different levels of representation of a solid model consistent.

Verification will be done at many different levels, This involves using mathematical theorem proving techniques to determine that the design correctly implements the specification. Tedious aspects of such techniques should be automated. When a designer completes the design of a component, he/she can verify that the implementation of that component meets its specification. When a set of components are interconnected, their specifications are composed and checked against the specification of the larger model.

An advantage of such a workstation is that it keeps all the different representation of a model consistent. This is accompanied by formally abstracting between different levels of description. When a level of description changes, the system should alert the user to the changes that are required at other levels to maintain consistency.

8.4 Final Thoughts

Finally, from a more general perspective, the work presented in this thesis has concentrated on the design process. If a fully formal integrated design methodology is to be developed - progressing from the early stages of concept development to the final implementation - then what has been presented here should aid the design, and in particular the specification and verification process. These phases in the design process are still some distance from the final manufacturing stage. As the verification technology matures and permeates both top level specifications and the lower level descriptions of implementations, formally verified systems in safety-related engineering applications should become a reality.

References

1. ACARD: Software: A Vital Key to UK Competitiveness, HMSO, 1986.
2. Agin G.J., and Binford T.O., Computer Descriptions of Curved Objects, IEEE Trans. Comp., Vol C-25, No.4, pp 439-449, April 1976.
3. Anderson L.R., Electric Machines and Transformers, Reston Publishing Co., 1981.
4. AQAP 1, NATO Quality Control System Requirements for Industry, September, 1985.
5. AQAP 13, NATO Software Quality Control System Requirements, September 1985.
6. Artin E., Introduction to Algebraic Topology, Columbus, Merrill, 1969.
7. Back R.J.R., Mannila H., A Semantic Approach to Program Modularity, Report C-1982-102, Department of Computer Science, University of Helsinki, 1982.
8. Back R.J.R., Wright, J. Von, Refinement Concepts Formalised in Higher Order Logic, in Proceedings of IFIP Working Conference on Programming Concepts and Methods, Broy M., Jones C.B. (eds), Amsterdam, Elsevier, 1990.
9. Baer A., Eastman C.M., and Henrion, M., Geometric Modelling: A Survey,, Computer Aided Design, 11, 1980, pp 253-272.

10. Barnhill R.E., Riesenfeld R.F. (eds), Computer Aided Geometric Design, NewYork, Academic Press, 1974.

11. Barker J.W., Mathematical Theory of Program Correctness, Prentice-Hall, 1981.

12. Barrow H.G., Proving the Correctness of Digital Hardware Designs, in VLSI Design, Vol.7, July 1984, pp 64-77.

13. Bergstra J.A. et al, Module Algebra, Report CS-R8617, Centre for Mathematics and Computer Science, Amsterdam, 1986.

14. Bevier W., et al, An Approach to Systems Verification, Journal of Automated Reasoning, Vol.5, No.4, November 1989.

15. Bochmann G.V., Hardware Verification with Temporal Logic: An Example. IEEE Transactions on Computers, C-32, Part 3, March 1982, pp 223-231.

16. Bohn W., Farn G., and Kahmann J., A Survey of Curve and Surface Models in CAGD,, Computer Aided Geometric Design, 1, 1984, pp 1-60.

17. Boyer R.S., Moore J.S., A Computational Logic Handbook, Boston, MA: Academic, 1988.

18. Boyse J.W., Preliminary Design for a Geometric Modeller, Report GMR-2768,Computer Science Dept., General Motors Research Ltd., July 1978.

19. Braid I.C., Three Systems for Shape Design and Representation - A Review, in Proc. CAM-I International CAM Seminar, CAM-I, England, April 1975, pp 60-67.

20. Braid I.C., Designing With Volumes, PhD Dissertation, University of Cambridge, Cambridge, 1973.

21. Bryant R.E., A Switch Level Simulation Model for Integrated Logic Circuits, PhD Thesis, Laboratory for Computer Science, MIT, March 1981.

22. BS 5750:1987, Guide to Quality Management and Quality System.

23. BS 5882:1980, Specification for a Total Quality Assurance Programme for Nuclear Power Plants.

24. Burstall R.M., Goguen J.A., The Semantic of Clear, A Specification Language, in Lecture Notes in Computer Science 86, Springer-Verlag, 1980, pp 292-332.

25. Camilleri A.J., Executing Behavioural Definitions in Higher-Order Logic, Report No. 140, Computer Laboratory, Cambridge University, February, 1988.

26. Camilleri A.J., Mechanising CSP Trace Theory in Higher Order Logic, IEEE Transactions on Software Engineering, Vol.16, No.9, September, 1990, pp 993-1004.

27. Camurati P., Prinetto P., Formal Verification of Hardware Correctness, IEEE Computer, Vol 21, No. 7, July 1988, pp 8-19.

28. Chiyokura H., Kimura F., Design of Solids With Free-Form Surfaces, Computer Graphics, Proc. of SIGGRAPH, 1983, 17, pp 289-198.

29. Church A., A Formulation of the Simple Theory of Types, The Journal of Symbolic Logic, 1940, Vol 5, pp 56-58.

30. Cohen B. et al, Specifications of Complex Systems, Addison-Wesley, 1986.

31. Cohn A., Correctness Properties of the Viper Block Model: The Second Level, Technical Report 134, University of Cambridge, May 1988.

32. Cohn A., The Notion of Proof in Hardware Verification, Journal of Automated Reasoning, Vol.5, May 1989, pp 127-139.

33. Cousineau G., Huet G., Paulson L., The ML Handbook, INRIA, 1986.

34. Craigen D., et al, m-EVES: A tool for Verifying Software, in Proceedings of 10th International Conference on Software Engineering, Los Alamitos, California, April, 1988, pp 324-333.

35. Cullyer W.J., Implementing Safety-Critical Systems: The Viper Microprocessor, in VLSI Specification, Verification and Synthesis, Birthwistle G., Subrahmanyam P.A. (eds.), Kluwer Academic Publishers, 1988, pp 1 -25.

36. Def Stan 00-16: Guide to the Achievement of Quality in Software, MOD.

37. Dhingra I.S., Formalising an Integrated Circuit Design Style in Higher-Order Logic, Ph.D. Thesis, Report No. 151, Computer Laboratory, Cambridge University, 1989.

38. Feinberg R., Modern Power Transformer Practice, Macmillan Press, 1979.

39. Fujita M. et al, Verification With Prolog and Temporal Logic, in Computer Hardware Description Languages and Their Applications, Uehara T. Barbacci M. (eds.), North-Holland, 1983, pp 103-114.

40. Goguen J.A., Tardo, J., An Introduction to OBJ: A language for Writing and Testing Algebraic Program Specification, in Software Specification Techniques, Gehani N., McGettrick A., (eds), Reading, MA: Addison-Wesley, 1979.

41. Goguen J.A., Burstall R.M., A Study in the Foundation of Programming Methodology: Specifications, Institutions, Charters and Parchments, in Category Theory and Computer Programming, Pitt D. et al (eds.), Lectures Notes in Computer Science, Springer-Verlag, 1986, pp 313-333.

42. Good D.I. et al, Principles of Proving Concurrent Program in Gypsy, Institute of Computer Science, Univerity of Texas, Austin, Technical Report, ICSCA-CMP-15, January, 1979.

43. Gordon M.J.C., Milner A.J., Wadsworth C.P., Edinburgh LCF : A Mechanized Logic of Computation, Springer Lecture Notes in Computer Science 78,

Springer Verlag, 1979.

44. Gordon M.J.C., LCF-LSM, Technical Report No.41, University of Cambridge, Computer Laboratory, 1981.

45. Gordon M.J.C., A Model of Register Transfer Systems With Application to Microcode and VLSI Correctness, Technical Report CSR-82-81, Department of Computer Science, University of Edinburgh, May 1982.

46. Gordon M.J.C., LCF-LSM : A System for Specifying and Verifying Hardware, Technical Report 41, Computer Laboratory, University of Cambridge, UK, 1983.

47. Gordon M.J.C., Proving a Computer Correct With the LCF-LSM Hardware Verification System, Technical Report 42, Computer Laboratory, University of Cambridge, UK, 1983.

48. Gordon M.J.C., and Herbert, J., A Formal Hardware Verification Methodology and Its Application to a Network Interface Chip, Technical Report 66, Computer Laboratory, University of Cambridge, UK, 1985.

49. Gondon M.J.C., HOL : A Machine Oriented Formulation of Higher Order Logic, Technical Report No.68, Computer Laboratory, University of Cambridge, Cambridge, UK, 1985.

50. Gordon M.J.C., HOL - A Proof Generating System for Higher-Order Logic, in VLSI Specification, Verification and Synthesis, Birtwistle G., Subrahmanyam P.A. (eds), Kluwer Academic Publishers, Boston, 1988, pp 73-128, Proceedings of the Hardware Verification Workshop, Calgary, Canada, 12-16, January, 1987.

51. Gordon M., Mechanizing Programming Logics in Higher-Order Logic, in Current Trends in Hardware Verification and Automated Theorem Proving, Birthwistle G. and Subrahmanyam P. (eds.), Springer-Verlag, New York, 1989, pp 387-439.

52. Hale R.S., Programming in Temporal Logic, Ph.D. Thesis, Report No. 173, Computer Laboratory, Cambridge University, July, 1989.

53. Hanna F.K., Daeche N., Specification and Verification Using Higher Order Logic : A Case Study, In Formal Aspects of VLSI Design : Proc. of the 1985 Edinburgh Conference on VLSI, Milne G.J., Subrahmanyam P.A., (eds), pp 179-213, North Holland, 1986.

54. Hanna F.K., and Daeche N., Specification and Verification of Digital Systems Using Higher Order Predicate Logic, IEEE Proceedings, Part E, September 1986, pp 242-254.

55. Hatcher W., The Logical Foundations of Mathematics, Pergamon Press, 1982.

56. Herbert J., Application of Formal Methods to Digital System Design, Ph.D. Dissertation, University of Cambridge, 1986.

57. Herbert J., Temporal Abstraction of Digital Designs, in The Fusion of Hardware Design and Verification, Milne G. (ed) , Procs. of the IFIP WG 10.2, International Working Conf., North-Holland, 1988, pp 1-25.

58. Hoare C.A., Communicating Sequential Processes, Prentice-Hall, 1985.

59. Hoare C.A.R. et al., Weakest Prespecifications, Technical Report PRG-44, Programming Research Group, Oxford University, 1985.

60. HSE PES: Programmable Electronic Systems in Safety-Related Applications, Vol.1. An introductory Guide, Vol.2. General Technical Guidelines, HMSO, 1987.

61. Hunt W.A., FM8510: A Verified Microprocessor, Ph.D. Thesis, Institute for Computer Science, University of Texas at Austin, 1986.

62. IEC 880: Software for Computers in Safety Systems of Nuclear Power Stations, International Electrotechnical Commission, IEC, 1986.

63. IEC SC65A WG9/10, Software for Computers in the Applications of Industrial Safety-Related Systems, Proposal for a Standard, International Electrot :chnical Commission, Subcommittee 65A, Working Group 9, 1989.

64. Inan K., Varaiya P., Finitely Recursive Process Models for Discrete Event Systems, IEEE Transactions on Automatic Control, Vol. 33, July 1988, pp 626-639.

65. Interim Defence Standard 00-31: Development of Safety Critical Software for Aircraft, MOD, 1987.

66. Interim Def Stan 00-55, Requirements For the Procurement of Safety Critical Software in Defence Equipment, Draft, Ministry of Defence, 1989.

67. Jones C.B., Systematic Development Using VDM, Prentice-Hall, 1986.

68. Joyce J., Formal Verification and Implementation of a Microprocessor, in VLSI Specification, Verification and Synthesis, Birtwistle G., Subrahmanyam P. (eds.), Kluwer Academic Publisher, 1988. pp 129-157.

69. Joyce J., Formal Specification and Verification of Asynchronous Processes in Higher-Order Logic, Report No. 136, Computer Laboratory, Cambridge University, June 1988.

70. Koymans R. et al, Compositional Semantics for Real-Time Distributed Computing, LNCS 193, Springer-Verlag, June 1985.

71. Kuratowski K., Mowtowski A., Set Theory, Amsterdam, North Holland Publishing Co., 1976.

72. Leeser M.E., Reasoning about the Function and Timing of Integrated Circuits with Prolog and Temporal Logic, Ph.D. Thesis, Computer Laboratory, Report No. 132, Computer Laboratory, Cambridge University, April 1988.

73. Leisenring A., Mathematical Logic and Hilbert's ϵ-Symbol, Macdonald & Co. Ltd., London, 1969.

74. Leveson N.G., Stolzy J.L., Safety Analysis Using Petri Nets, IEEE Transaction on Software Engineering, SE-13(3), March 1987, pp 386-397.

75. Loewenstein P., Reasoning about State Machines in Higher-Order Logic, in Specification, Verification and Synthesis: Mathematical Aspects, Goos G. et al (eds.), Lecture Notes in Computer Science, Vol 408, 1990, Springer-Verlag, pp 67-89.

76. Mallgren W.R., Formal Specification of Graphic Data Types, ACM Transactions on Programming Languages and Systems, Vol.4, October 1982, pp 687-710.

77. Manna Z., Pnueli A., The Anchored Version of the Temporal Framework. In Models of Concurrency: Linear, Branching and Partial Orders, Barker J. W. et al (eds.), LNCS. Springer-Verlag, 1989.

78. Markowski, G., Wesley M.A., Fleshing Out Wire Frame, IBM Journal of Research and Development, 1980, 24, pp 582-597.

79. Melham T., Abstraction Mechanisms for Hardware Verification, Report No. 103, Computer Laboratory, Cambridge University, January 1987.

80. Middleditch A.E. et al, Blend Surfaces for Set Theoretic Volume Modelling, Computer Graphics, Proc. of SIGGRAPH 85, 1985, 19, pp 161-170.

81. Milne G.J., Milner R., Concurrent Processes and Their Syntax, Journal of the ACM, Vol.26, April 1979.

82. Milne G.J., CIRCAL: A Calculus for Circuit Description, Integration, Vol.1, Part 2-3, October 1983, pp 121-160.

83. Milne G.J., CIRCAL and the Representation of Communication, Concurrency and Time, Technical Report CSR-151-83, Department of Computer Science, University of Edinburgh, Novemebr 1983.

84. Milne R., Design Transformation and Chip Planning, In Milne G., Subrahmanyam P.A. (eds.), Formal Aspects of VLSI Design, North-Holland, 1986, pp 23-43.

85. Milner R., A Calculus of Communicating Systems, in Lectures Notes in Computer Science, Vol.92, Springer Verlag, 1980.

86. Milner R., Calculi for Synchrony and Asynchrony, Technical Report CSR-104-82, Department of Computer Science, University of Edinburgh, August 1982.

87. Mortenson M.E., Geometric Modelling, NewYork, John Wiley, 1985.

88. Moszkowski B.C., Reasoning about Digital Circuits, Ph.D. Thesis, Report CS-83-970, Dept. of Computer Science, Stanford University, 1983.

89. Moszkowski B.C., A Temporal Logic for Multilevel Reasoning about Hardware, IEEE Computer Vol. 18, No. 2, February 1985, pp 10-19.

90. NES 620, Requirements for Software for Use With Digital Computers.

91. Paulson L., A Higher-Order Implementation of Rewriting, Science of Computer Programming, Volume 3, Part 2, August 1983, pp 119-149.

92. Paulson L., Tactics and Tacticals in Cambridge LCF, Technical Report 39, Computer Laboratory, University of Cambridge, UK, 1983.

93. Paulson L., Logic and Computation, Cambridge University Press, 1987.

94. Pickett M.S., Boyse J.W., Solid Modelling by Computers, NewYork, Plenum Press, 1984.

95. Quirk W., Verification and Validation of Real time Software, Springer-Verlag, Berlin, 1985.

96. Reed G.M., A Timed Model for Communicating Sequential Processes, in Proceedings ICALP 86, Springer-Verlag, LNCS 226, 1986.

97. Requicha A.A., Tilove R.B., General Topology of Closed Regular Sets, Tech. Memo. No.27, Production Automation Project, University of Rochester, June 1978.

98. RTCA/DO-178: Software Considerations in Airborne Systems and Equipment Certification, Published by Radio Technical Commission for Aeronautics, Washington, DC, November, 1985.

99. Sabin M.A., Some Negative Results in N sided Patches, Computer Aided Design, 18, 1986, pp 38-44.

100. Sanella D.T., Wirsing M., A Kernel Language for Algebraic Specification and Implementations, Report CSR-131-83, Department of Computer Science, University of Edinburgh, 1983.

101. Sheeran M., μFP, An Algebraic VLSI Design Language, PhD Thesis, University of Oxford, November 1983.

102. Simmons G.P., Introduction to Topology and Modern Analysis, NewYork, Mc GrawHill, 1968.

103. Sokolowski S., Soundness of Hoare's Logic: An Automatic Proof Using LCF, ACM Transactions on Programming Languages and Systems, Vol.9, 1987, pp 100-120.

104. Spivey J.M., Understanding Z, Cambridge University Press, 1988.

105. STARTS Purchasers Handbook - Procuring Software-Based Systems, NCC Ltd. Manchester, 1989.

106. Subrahmanyam P.A., Synapse: An Expert System for VLSI Design, Computer, 1986, pp 78-89.

107. Tran S., Guidelines for Hazard Analysis, Technical Report No. 620/010/90, T&N Technology Ltd., July 1989.

108. Tran S., Standard for the Procurement of Electronic Systems in Vehicle, Technical Report No. 621/010/90, T&N Technology Ltd, Sept 1989.

109. Tran S., Cullyer J., Hines E., Marks K., The Development of a High Quality Software Design Methodology for Automotive Applications, IEE Conf. on Safety Critical Software in Vehicle and Traffic Control, London, Feb 1990.

110. Tran S., Cullyer J., Hines E., Marks K., Formal Methods in Automotive Applications, in the Proc. of the 22nd ISATA International Conference, Florence, Italy, May 1990.

111. Tran S., Cullyer J., Hines E., Marks K., On the Development of Formal Methods Based Software Design Methodology for Automotive Applications, Microprocessor and Microsystems, 14(5), June 1990, pp 320-323.

112. Traub N., A Lisp Based CIRCAL Environment, Technical Report CSR-152-83, Department of Computer Science, University of Edinburgh, November 1983.

113. Traub N., A Formal Approach to Hardware Analysis, Technical Report CST-43-87, Department of Computer Science, University of Edinburgh, March 1987.

114. Van Eijk P.H.J., Visser C.A., Diaz M. (eds.), The Formal Description Technique LOTOS, Elsevier North-Holland, NewYork, 1989.

115. Wikstrom A., Functional Programming Using Standard ML, Prentice Hall International, 1987.

116. Winskel G., A Compositional Model of MOS Circuits, in VLSI Specification, Verification and Synthesis, Birthwistle G., Subrahmanyam P.A. (eds.), Kluwer Academic Publishers, 1988, pp 323-347.

117. Zave P., An Operational Approach to Requirements Specification for Embedded Systems, IEEE Transactions on Software Engineering, Vol. 8, Part 3, May 1982, pp 250-269.

APPLICATIONS OF FORMAL METHODS IN ENGINEERING

Volume 2 of 2

Sang Cong Tran
B.Eng (Hons.), M.Phil (Eng.)

A thesis submitted for the degree of
Doctor of Philosophy in the Engineering Department
University of Warwick
December 1991

Acknowledgement

During the time of this project, it is pleasing to note the number of people who have given their help and support. Thanks are due to my supervisor Dr. Evor Hines. He diligently read through the various drafts of this thesis and made valuable written comments. His insistence on clear writing has set a standard which I will always try to aim for.

In addition, I would like to express my sincerest thanks to Professor John Cullyer. This work could never have taken shape without his continued guidance and support. His patience during my earlier days and his ever optimistic manner has brought me through some of the most difficult times. He made valuable suggestions on earlier drafts of this thesis, and was the source of many excellent discussions.

This work could not have started without the financial support of T&N Technology Limited, Goodyear Transformers Limited and the Science and Engineering Research Council. The technical advice and individual guidance received while working there greatly help in this project. Of the innumerable people who helped, I must explicitly acknowledge Kevin Marks at T&N Technology Ltd. and Ken Frewin at Goodyear Transformers Ltd., who were the source of many stimulating discussions.

Thanks are also due to colleagues who have been around at various times to ask questions or answer mine. I would particularly like to thank Wai Wong for help at various times with HOL theorems and tactics and in response to my persistent questions about LATEX typesetting.

This work would not have been completed without the HOL system. Thanks are due to the hardware verification group at Cambridge University. I am especially grateful to Dr. Mike Gordon for leading the way in hardware verification and giving advice when it was most needed.

Abstract

The main idea presented in this thesis is to propose and justify a general framework for the development of safety-related systems based on a selection of criticality and the required level of integrity. We show that formal methods can be practically and consistently introduced into the system design lifecycle without incurring excessive development cost.

An insight into the process of generating and validating a formal specification from an engineering point of view is illustrated, in conjunction with formal definitions of specification models, safety criteria and risk assessments. Engineering specifications are classified into two main classes of systems, memoryless and memory bearing systems. Heuristic approaches for specification generation and validation of these systems are presented and discussed with a brief summary of currently available formal systems and their supporting tools.

It is further shown that to efficiently address different aspects of real-world problems, the concept of embedding one logic within another mechanised logic, in order to provide mechanical support for proofs and reasoning, is practical. A temporal logic framework, which is embedded in Higher Order Logic, is used to verify and validate the design of a real-time system. Formal definitions and properties of temporal operators are defined in HOL and real-time concepts such as timing marker, interrupt and timeout are presented. A second major case study is presented on the specification a solid model for mechanical parts. This work discusses the modelling theory with set theoretic topology and Boolean operations. The theory is used to specify the mechanical properties of large distribution transformers. Associated mechanical properties such as volumetric operations are also discussed.

Declaration

This dissertation is the result of my own work and, unless otherwise stated in the text, includes nothing which is the outcome of work done in collaboration. No part of this dissertation has already been, or is currently being, submitted for any degree, diploma or other qualification at any other university.

Contents

Annex A: A Formal Solid Modelling System

A.1 Introduction

In order to support the solid modelling theory described in chapter 7, a simple solid modeller has been implemented. The program was written using a subset of Pascal into which the HOL source can be translated directly. The system includes a set of pre-constructed primitives and can support a number of geometric operations.

Basically, the program is a batch based solid modeller. It reads commands either from keyboard or file and executes the commands in order to create new 3-dimensional objects. It is based on the HOL theory of solid modelling which is a formal representation scheme in which complex solid may be defined as combinations of primitive solid *building blocks* via the regularised set operators. The architecture of the system is shown in Fig.A.1.

A.2 System Descriptions

The definition of objects in FSMS consists of two types of statements: definitional statements and commands in imperative format. For instance,

```
(1)   c1=cylin(vector(1500,1375,900),vector(0,0,1500),300);
(2)   c2=cylin(vector(1500,1375,900),vector(0,0,1400),400);
(3)   coil1=c2-c1;
```

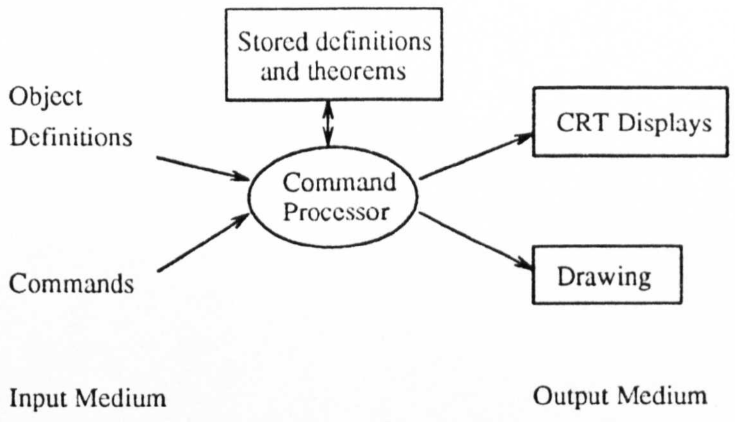


Fig. A.1 System architecture

In this example, statements (1) and (2) define primitive solids, which are cylindrical objects in this case. Composite solids are defined via regularised set operations in statement (3) [1].

A.3 Data Types

Table A.1 shows data types supported by FSMS. To ensure the consistencies and validity of the formal theory, the system types are defined in a similar way as those specified in the HOL theory. There are three main types, From these generic

Type	Value
ObjectType	Solid.
NumericType	Integer type.
ObjListType	list of (any of the above type) objects.

Table A.1 System data types

system types, an entity type may be determined by its symbolic name. In the example presented above, c1 is a symbolic name of the cylinder specified by its appropriate dimensions. However, names in FSMS are not *variables*, thus only

[1]In FSMS, regularised set operations have in the same format as Boolean operations.

one value may be associated to each name in the definition.

A.4 Primitive Solids

The system supports two primitive solids: blocks and three versions of cylinders, whose axes must be parallel to the (X, Y, Z) master coordinate system. These are defined as follows,

```
BOX(Point, Dx, Dy, Dz)
```

defines a rectangular solid block at a base position of `Point` and `Dx`, `Dy`, `Dz` as its dimensions. Note that negative dimensions are not allowed. The basic element in FSMS is defined as `Point` which is a 3-tupple of the form (x,y,z) where x, y, z are its real coordinates.

In a similar manner, a cylinder is presented by a formula

```
CYLIN (Centre, Direction, Radius)
```

where the cylinder is a geometrical solid object which has `Centre` as its base centre, and `Radius` as its radial dimension. `Direction` is presented as a tupple of 3-element, ie. a vector of the form (dx,dy,dz) where the magnitude of each element represents the cylinder height. Therefore, the formula can be used to present cylindrical objects in all three dimensions.

```
X direction :  CYLIN (Centre, (10,0,0), Radius)
Y direction :  CYLIN (Centre, (0,10,0), Radius)
Z direction :  CYLIN (Centre, (0,0,10), Radius)
```

These formulae represents cylinder of the same size in X, Y, Z direction respectively.

A.5 System Operators

Three types of operators are available in FSMS,

- Positional operators (rigid motion)

  The translational function `TRANS (Distance)` translates a solid by the specified `Distance` argument. Since HOL theory does not support rational arguments, therefore to preserve the validity, rotational operations are not implemented in FSMS.

- Combinational operators on solid

  The three regularised set-theoretical operators UNION, INTER and DIFF are included. Combinational operators are written in infix form. Precedence between operators is defined by bracketing structures. It should be noted that, regularised set operators are *general*, and may be applied to primitives as well as any parts defined in FSMS. In this way, complex solid objects can be hierarchically defined without the risk of introducing inconsistencies at any stage in the design since each step of the development is verified by the system.

- Arithmetic operators

  The system supports the usual arithmetic operators such as +, -, * which operate on integers. Division operation however, is not included.

A.6 Definitional Statements

There are two kinds of definitional statements. These are object definitional statements, and value assignment statements. Object definitional statements are of the form

```
<name> = <obj>
```

where `<obj>` is any expression that has a *solid* value. Statements (1) and (2) in the earlier example are definitional statements. Value assignment statements associate values with numbers, for example,

```
x = 5
```

The order of definitional statements in FSMS is important, a name can not be used before it has been assigned a value.

A.7 Commands

A number of useful commands are included to support the design environment. Commands may appear anywhere in a FSMS definition and are executed immediately if possible. FSMS has three types of commands,

- Utility commands, eg. to `INCLUDE` and `LOAD` files containing definitions.
- Editing commands: editing can be done interactively at the system prompt in a line editing format, or in screen editing format of the user's specified editor. A setup file allows user to specify his/her chosen editor before starting FSMS.
- Graphic commands: FSMS graphic commands were implemented in a way to relieve users of the burden of specifying the multitude of parameters needed to ensure that displays fit on the screen. The basic syntax is

  ```
  VIEW (ObjectList, ClearWindow)
  ```

  The `VIEW` command takes a list of objects and draw on the screen. The Boolean value `ClearWindow` specifies whether the graphic screen is blanked before drawing. In fact, function `VIEW` is described in exactly the same way as the HOL function `BUILD` specified in chapter 7.

A.8 Limitations

There are a number of restrictions on the operations between objects.

- No intersection of co-planar objects is allowed. One can always overcome its model by moving one of the operands in the boolean operation by a small amount.

- If the intersection of two objects falls exactly on their boundaries, the system will alarm that the two object do not intersect at all, since their intersection results in dangling faces.
- Intersection that results with a point or a line will probably cause wrong propagation of the inner and outer part of one object relative to the other. There are bugs in the BOUNDARY evaluation algorithms which remain intact.

A.9 Comments

This modelling system is far from been practical. The main aim of implementing FSMS is to give a visual demonstrations of the validity of the modelling theory which is, due to its generality, of a high level of abstraction. However, it may serve as a pointer which illustrates the feasibility of combining formal techniques to real-world problems.

A.10 Design Example

An FSMS source for modelling a distribution transformer is included as follows,

```
# Dimension in mm

# CORE = (((BOX(1375,1000,750)(1757,750,2000) REG_UNION
#           BOX(1500,1250,500)(250,250,500) REG_UNION
#           BOX(2750,1250,500)(250,250,500)) REG_DIFF
#           BOX(1625,500,1000)(500,2000,1500)) REG_DIFF
#           BOX(2375,500,1000),(500,2000,1500))

core=box(vector(1375,1000,750),1750,750,2000);
b1=box(vector(1500,1250,500),250,250,500);
b2=box(vector(2750,1250,500),250,250,500);
core=core+b1;
core=core+b2;
```

```
free(b1);
free(b2);
b1=box(vector(1625,500,1000),500,2000,1500);
b2=box(vector(2375,500,1000),500,2000,1500);
core=core-b1;
core=core-b2;
free(b1);
free(b2);
view(list(core),off);
free(core);

# COIL = (CYLINDER_Z(1500,1375,900)(1500,300) REG_DIFF
#         CYLINDER_Z(1500,1375,900)(1400,400)) REG_UNION
#        (CYLINDER_Z(2250,1375,900)(1500,300) REG_DIFF
#         CYLINDER_Z(2250,1375,900)(1400,400)) REG_UNION
#        (CYLINDER_Z(3000,1375,900)(1500,300) REG_DIFF
#         CYLINDER_Z(3000,1375,900)(1400,400))

c1=cylin(vector(1500,1375,900),vector(0,0,1500),300);
c2=cylin(vector(1500,1375,900),vector(0,0,1400),400);
coil1=c2-c1;
free(c1);
free(c2);
view(list(coil1),off);
free(coil1);
c1=cylin(vector(2250,1375,900),vector(0,0,1500),300);
c2=cylin(vector(2250,1375,900),vector(0,0,1400),400);
coil2=c2-c1;
free(c1);
free(c2);
view(list(coil2),off);
free(coil2);
c1=cylin(vector(3000,1375,900),vector(0,0,1500),300);
c2=cylin(vector(3000,1375,900),vector(0,0,1400),400);
```

```
coil3=c2-c1;
free(c1);
free(c2);
view(list(coil3),off);
free(coil3);

# LUGS = (BOX(750,1500,3600)(250,100,400) REG_DIFF
#          CYLINDER_Y(875,1400,3800)(600,75)) REG_UNION
#         (BOX(3500,1500,3600)(250,100,400) REG_DIFF
#          CYLINDER_Y(3625,1400,3800)(600,75))

lug=box(vector(750,1500,3600),250,100,400);
c=cylin(vector(875,1400,3800),vector(0,600,0),75);
lug = lug-c;
view(list(lug),off);
free(lug);
free(c);
lug=box(vector(3500,1500,3600),250,100,400);
c=cylin(vector(3625,1400,3800),vector(0,600,0),75);
lug = lug-c;
view(list(lug),off);
free(lug);
free(c);

# TANK = BOX(1000,500,500)(2500,2000,3500)

tank = box(vector(1000,500,500),2500,2000,3500);
view(list(tank),off);
free(tank);

# COOLING = CYLINDER_X(0,1400,4250)(1500,400) REG_UNION
#           CYLINDER_Z(1250,1700,3900)(300,100)

cooling=cylin(vector(0,1400,4250),vector(1500,0,0),400);
```

```
c=cylin(vector(1250,1700,3900),vector(0,0,300),100);
cooling = cooling+c;
view(list(cooling),off);
free(c);
free(cooling);

# CABLE = BOX(1875,2500,2750)(750,500,750) REG_UNION
#         BOX(2000,2625,2500)(500,250,500) REG_UNION
#         CYLINDER_Y(2000,2750,3000)(350,50) REG_UNION
#         CYLINDER_Y(2250,2750,3000)(350,50) REG_UNION
#         CYLINDER_Y(2500,2750,3000)(350,50)

b1 = box(vector(1875,2500,2750),750,500,750);
b2 = box(vector(2000,2625,2500),500,250,500);
cable=b1+b2;
free(b1);
free(b2);
c1=cylin(vector(2000,2750,3000),vector(0,350,0),50);
c2=cylin(vector(2250,2750,3000),vector(0,350,0),50);
c3=cylin(vector(2500,2750,3000),vector(0,350,0),50);
cable = cable+c1;
cable = cable+c2;
cable = cable+c3;
free(c1);
free(c2);
free(c3);
color(cable,red);
view(list(cable),off);
free(cable);

# BUSHING = CYLINDER_Z(3250,1500,4100)(50,150) REG_UNION
#           CYLINDER_Z(3250,1500,4250)(50,150) REG_UNION
#           CYLINDER_Z(3250,1500,4400)(50,150) REG_UNION
#           CYLINDER_Z(3250,1500,4000)(650,50) REG_UNION
```

```
#
#            CYLINDER_Z(2850,1500,4100)(50,150) REG_UNION
#            CYLINDER_Z(2850,1500,4250)(50,150) REG_UNION
#            CYLINDER_Z(2850,1500,4400)(50,150) REG_UNION
#            CYLINDER_Z(2850,1500,4000)(650,50) REG_UNION
#
#            CYLINDER_Z(2450,1500,4100)(50,150) REG_UNION
#            CYLINDER_Z(2450,1500,4250)(50,150) REG_UNION
#            CYLINDER_Z(2450,1500,4400)(50,150) REG_UNION
#            CYLINDER_Z(2450,1500,4000)(650,50) REG_UNION

b1=cylin(vector(3250,1500,4100),vector(0,0,50),150);
b2=cylin(vector(3250,1500,4250),vector(0,0,50),150);
b3=cylin(vector(3250,1500,4400),vector(0,0,50),150);
col=cylin(vector(3250,1500,4000),vector(0,0,650),50);
bushing1=b1+b2;
bushing1=bushing1+b3;
bushing1=bushing1+col;
free(col);
free(b1);
free(b2);
free(b3);
view(list(bushing1),off);
free(bushing1);
b1=cylin(vector(2850,1500,4100),vector(0,0,50),150);
b2=cylin(vector(2850,1500,4250),vector(0,0,50),150);
b3=cylin(vector(2850,1500,4400),vector(0,0,50),150);
col=cylin(vector(2850,1500,4000),vector(0,0,650),50);
bushing2=b1+b2;
bushing2=bushing2+b3;
bushing2=bushing2+col;
free(col);
free(b1);
free(b2);
free(b3);
view(list(bushing2),off);
free(bushing2);

b1=cylin(vector(2450,1500,4100),vector(0,0,50),150);
b2=cylin(vector(2450,1500,4250),vector(0,0,50),150);
b3=cylin(vector(2450,1500,4400),vector(0,0,50),150);
col=cylin(vector(2450,1500,4000),vector(0,0,650),50);
bushing3=b1+b2;
bushing3=bushing3+b3;
bushing3=bushing3+col;
free(col);
free(b1);
free(b2);
free(b3);
view(list(bushing3),off);
free(bushing3);

# RADIATOR = BOX(850,750,750)(50,1500,2500) REG_UNION
#            BOX(700,750,750)(50,1500,2500) REG_UNION
#            BOX(550,750,750)(50,1500,2500) REG_UNION
#            BOX(400,750,750)(50,1500,2500) REG_UNION
#            BOX(250,750,750)(50,1500,2500) REG_UNION
#            BOX(100,750,750)(50,1500,2500) REG_UNION

#            BOX(3650,750,750)(50,1500,2500) REG_UNION
#            BOX(3800,750,750)(50,1500,2500) REG_UNION
#            BOX(3950,750,750)(50,1500,2500) REG_UNION
#            BOX(4100,750,750)(50,1500,2500) REG_UNION
#            BOX(4250,750,750)(50,1500,2500) REG_UNION
#            BOX(4400,750,750)(50,1500,2500) REG_UNION

panel1=box(vector(850,750,750),50,1500,2500);
panel2=box(vector(700,750,750),50,1500,2500);
```

```
panel3=box(vector(550,750,750),50,1500,2500);
panel4=box(vector(400,750,750),50,1500,2500);
panel5=box(vector(250,750,750),50,1500,2500);
panel6=box(vector(100,750,750),50,1500,2500);
radiator1=panel1+panel2;
radiator1=radiator1+panel3;
radiator1=radiator1+panel4;
radiator1=radiator1+panel5;
radiator1=radiator1+panel6;
free(tube2);
free(tube1);
free(panel1);
free(panel2);
free(panel3);
free(panel4);
free(panel5);
free(panel6);
view(list(radiator1),on);
free(radiator1);
panel1=box(vector(3650,750,750),50,1500,2500);
panel2=box(vector(3800,750,750),50,1500,2500);
panel3=box(vector(3950,750,750),50,1500,2500);
panel4=box(vector(4100,750,750),50,1500,2500);
panel5=box(vector(4250,750,750),50,1500,2500);
panel6=box(vector(4400,750,750),50,1500,2500);
radiator2=panel1+panel2;
radiator2=radiator2+panel3;
radiator2=radiator2+panel4;
radiator2=radiator2+panel5;
radiator2=radiator2+panel6;
free(tube2);
free(tube1);
free(panel1);
free(panel2);
free(panel3);
free(panel4);
free(panel5);
free(panel6);
view(list(radiator2),off);
free(radiator2);

# BASE = (BOX(500,500,0),3500,2000,500) REG_DIFF
#         CYLINDER_Y(750,0,250)(4000,50)) REG_DIFF
#         CYLINDER_Y(3750,0,250)(4000,50)

bottom = box(vector(500,500,0),3500,2000,500);
hole1 = cylin(vector(750,0,250),vector(0,4000,0),50);
hole2 = cylin(vector(3750,0,250),vector(0,4000,0),50);
base = bottom - hole1;
base = base - hole2;
free(bottom);
free(hole1);
pfree(hole2);
view(list(base),off);
free(base);
```

Annex B: Temporal Definitions and Theorems

B.1 Definitions

And_DEF	$\vdash \forall x\ y.\ x \ \mathsf{And}\ y = (\lambda t.\ x\ t\ \wedge\ y\ t)$
Or_DEF	$\vdash \forall x\ y.\ x \ \mathsf{Or}\ y = (\lambda t.\ x\ t\ \vee\ y\ t)$
Not_DEF	$\vdash \forall x.\ \mathsf{Not}\ x = (\lambda t.\ \neg x\ t)$
Imp_DEF	$\vdash \forall x\ y.\ x \longrightarrow y = (\lambda t.\ x\ t\ \supset\ y\ t)$
Xor2_DEF	$\vdash \forall x\ y.\ \mathsf{Xor2}\ (x, y) = (\lambda t.\ (x\ t\ \vee\ y\ t)\ \wedge\ \neg(x\ t\ \wedge\ y\ t))$
Xor3_DEF	$\vdash \forall x\ y\ z.\ \mathsf{Xor3}\ (x, y, z) = (\lambda t.\ (x\ t\ \vee\ y\ t\ \vee\ z\ t)$ $\wedge\ \neg(x\ t\ \wedge\ y\ t\ \vee\ y\ t\ \wedge\ z\ t\ \vee\ z\ t\ \wedge\ x\ t))$
True_DEF	$\vdash \mathsf{True} = (\lambda t.\ \mathsf{T})$
False_DEF	$\vdash \mathsf{False} = (\lambda t.\ \mathsf{F})$
Next_DEF	$\vdash \forall p.\ \mathsf{Next}\ p = (\lambda t.\ p\ (\mathsf{SUC}\ t))$
Until_DEF	$\vdash \forall p\ q.\ p\ \mathsf{Until}\ q = (\lambda t.\ (\exists t_1.\ t \leq t_1\ \wedge\ q\ t_1\ \wedge$ $(\forall t_2.\ t \leq t_2\ \wedge\ t_2 < t_1\ \supset\ p\ t_2)))$
Sometime_DEF	$\vdash \forall p.\ \mathsf{Sometime}\ p = (\lambda t.\ (\exists t'.\ t \leq t'\ \wedge\ p\ t'))$
Always_DEF	$\vdash \forall p.\ \mathsf{Always}\ p = (\lambda t.\ (\forall t'.\ t \leq t'\ \supset\ p\ t'))$
Eq_DEF	$\vdash \forall f\ c.\ f\ \mathsf{Eq}\ c = (\lambda t.\ f\ t = c)$
Equiv_DEF	$\vdash \forall p\ q.\ p\ \mathsf{Equiv}\ q = (\lambda t.\ p\ t = q\ t)$
First_DEF	$\vdash \forall p\ t.\ \mathsf{First}\ p\ t = p\ t\ \wedge\ (\forall t'.\ t' < t\ \supset\ \neg p\ t')$
Last_DEF	$\vdash \forall p\ t_1\ t_2.\ \mathsf{Last}\ p\ (t_1 . t_2) = (\forall t.\ t_1 < t\ \wedge\ t < t_2\ \supset\ \neg p\ t)\ \wedge\ p\ t_2$

Fin_DEF $\vdash \forall p\, t_1\, t_2.\, \mathsf{Fin}\ p\ (t_1, t_2) = t_1 < t_2\ \wedge\ p\ t_2$

Keep_DEF $\vdash \forall p\, t_1\, t_2.\, \mathsf{Keep}\ p\ (t_1, t_2) = t_1 < t_2\ \wedge\ (\forall t.\, t_1 < t\ \wedge\ t < t_2\ \supset\ p\ t)$

Length_DEF $\vdash (\forall p.\, \mathsf{Length}\ p\ 0 = (\lambda t.\, p\ t))\ \wedge$
$(\forall p\ n.\, \mathsf{Length}\ p\ (\mathsf{SUC}\ n) = (\lambda t.\, \mathsf{Length}\ p\ n\ t\ \wedge\ \neg p\ (\mathsf{SUC}\ (n + t))))$

Len_DEF $\vdash \forall p\ n.\, \mathsf{Len}\ p\ n = (\lambda t'.\, \mathsf{Length}\ p\ n\ (t' - n))$

Mapped_DEF $\vdash (\forall p.\, \mathsf{Mapped}\ p\ 0 = (\lambda t.\, \mathsf{First}\ p\ t))\ \wedge$
$(\forall p\ n.\, \mathsf{Mapped}\ p\ (\mathsf{SUC}\ n) = (\lambda t.\, (\exists t'.\, \mathsf{Mapped}\ p\ n\ t'\ \wedge\ \mathsf{Last}\ p\ (t', t))))$

Proj_DEF $\vdash \forall p\ n.\, p\ \mathsf{Proj}\ n = (\varepsilon t.\, \mathsf{Mapped}\ p\ n\ t)$

Valid_DEF $\vdash \forall p.\, \mathsf{Valid}\ p = (\forall t.\, p\ t)$

When_DEF $\vdash \forall p\ b.\, p\ \mathsf{When}\ b = (\forall n.\, b\ (p\ \mathsf{Proj}\ n))$

Upon_DEF $\vdash \forall p\ b.\, p\ \mathsf{Upon}\ b = (\lambda p'\ t.\, (p\ \mathsf{When}\ b \Rightarrow p'\ t \mid p\ t))$

Timeout_DEF

$\vdash (\forall b.\, \mathsf{Timeout}\ 0\ b = \mathsf{T})\ \wedge\ (\forall t\ b.\, \mathsf{Timeout}\ (\mathsf{SUC}\ t)\ b = (b \Rightarrow \mathsf{F} \mid \mathsf{Timeout}\ t\ b))$

Within_DEF $\vdash \forall b\ p.\, b\ \mathsf{Within}\ p = (\lambda t.\, \neg(p\ t\ \wedge\ \mathsf{Sometime}\ b\ t))$

B.2 Theorems

LESSEQ $\vdash \forall a\ b.\, \neg(a < b)\ \supset\ b \leq a$

SUC_LESS_SUC $\vdash \forall t\ t'.\, \mathsf{SUC}\ t \leq t'\ \supset\ \mathsf{SUC}\ t \leq \mathsf{SUC}\ t'$

NOT_EQ_REFL $\vdash \forall a\ b.\, (\neg a = b) = (a = \neg b)$

IMP_AND $\vdash \forall a\ b\ c.\, a\ \supset\ (b\ \supset\ c) = a\ \wedge\ b\ \supset\ c$

LESS_EQ_TRANS $\vdash \forall a\ b\ c.\, a \leq b\ \wedge\ b \leq c\ \supset\ a \leq c$

IMP_CONTRAPOS $\vdash \forall a\ b.\, a\ \supset\ b = \neg b\ \supset\ \neg a$

IMP_SELF $\vdash \forall a\ b.\, a\ n\ \supset\ (b\ n\ \supset\ a\ n)$

lemma1 $\vdash \forall x.\, x\ \mathsf{And}\ x = x$

lemma2 $\vdash \forall x\ y.\, x\ \mathsf{And}\ y = y\ \mathsf{And}\ x$

lemma3 $\vdash \forall x\ y\ z.\, (x\ \mathsf{And}\ y)\ \mathsf{And}\ z = x\ \mathsf{And}\ (y\ \mathsf{And}\ z)$

lemma4 $\vdash \forall x.\, x\ \mathsf{And}\ \mathsf{True} = x$

`lemma5`	$\vdash \forall x.\, x \text{ And False} = \text{False}$
`lemma6`	$\vdash \forall x.\, x \text{ Or } x = x$
`lemma7`	$\vdash \forall x\ y.\, x \text{ Or } y = y \text{ Or } x$
`lemma8`	$\vdash \forall x\ y\ z.\, (x \text{ Or } y) \text{ Or } z = x \text{ Or } (y \text{ Or } z)$
`lemma9`	$\vdash \forall x.\, x \text{ Or True} = \text{True}$
`lemma10`	$\vdash \forall x.\, x \text{ Or False} = x$
`lemma11`	$\vdash \forall x.\, \text{Xor2}\,(x, x) = \text{False}$
`lemma12`	$\vdash \forall x\ y.\, \text{Xor2}\,(x, y) = \text{Xor2}\,(y, x)$
`lemma13`	$\vdash \forall x.\, \text{Xor2}\,(x, \text{True}) = \text{Not}\ x$
`lemma14`	$\vdash \forall x.\, \text{Xor2}\,(x, y) = \text{Xor3}\,(x, y, \text{False})$
`lemma15`	$\vdash (\text{Not True} = \text{False}) \land (\text{Not False} = \text{True})$
`lemma16`	$\vdash \forall p\ t_1\ t_2.\, t_1 < t_2 \land \text{Last}\ p\,(t_1, t_2) =$ $\text{Keep}\,(\text{Not}\ p)\,(t_1, t_2) \land \text{Fin}\ p\,(t_1, t_2)$
`lemma17`	$\vdash \forall p.\, \text{Sometime}\ p = \text{True Until}\ p$
`lemma18`	$\vdash \forall p.\, \text{Always}\ p = \text{Not}\,(\text{Sometime}\,(\text{Not}\ p))$
`lemma19`	$\vdash \forall p.\, \text{Valid}\,(p \longrightarrow \text{Sometime}\ p)$
`lemma20`	$\vdash \forall p.\, \text{Valid}\,(\text{Next}\ p \longrightarrow \text{Sometime}\ p)$
`lemma21`	$\vdash \forall p.\, \text{Sometime}\,(\text{Not}\ p) = \text{Not}\,(\text{Always}\ p)$
`lemma22`	$\vdash \forall p\ q.\, \text{Valid}\,(\text{Always}\,(p \longrightarrow q) \longrightarrow (\text{Sometime}\ p \longrightarrow \text{Sometime}\ q))$
`lemma23`	$\vdash \forall p\ q.\, \text{Always}\,(p \text{ And } q) = \text{Always}\ p \text{ And Always}\ q$
`lemma24`	$\vdash \forall p\ q.\, \text{Sometime}\,(p \text{ Or } q) = \text{Sometime}\ p \text{ Or Sometime}\ q$
`lemma25`	$\vdash \forall p\ q.\, \text{Valid}\,((\text{Always}\ p \text{ And Always}\ q) \longrightarrow \text{Always}\,(p \text{ Or } q))$
`lemma26`	$\vdash \forall p\ q.\, \text{Valid}\,(\text{Sometime}\,(p \text{ And } q) \longrightarrow (\text{Sometime}\ p \text{ And Sometime}\ q))$
`lemma27`	$\vdash \forall p\ q.\, \text{Next}\,(p \text{ And } q) = \text{Next}\ p \text{ And Next}\ q$
`lemma28`	$\vdash \forall p\ q.\, \text{Next}\,(p \text{ Or } q) = \text{Next}\ p \text{ Or Next}\ q$
`lemma29`	$\vdash \forall p\ q.\, \text{Next}\,(p \longrightarrow q) = \text{Next}\ p \longrightarrow \text{Next}\ q$

`lemma30`	$\vdash \forall p\ q.\ \text{Valid}\ ((\text{Always}\ p\ \text{And Sometime}\ q) \longrightarrow (p\ \text{Until}\ q))$
`lemma31`	$\vdash \forall p.\ \text{Valid}\ (\text{Sometime}\ (\text{Next}\ p) \longrightarrow \text{Next}\ (\text{Sometime}\ p))$
`lemma32`	$\vdash \forall p.\ \text{Valid}\ (\text{Always}\ (\text{Sometime}\ (\text{Always}\ p)) \longrightarrow \text{Sometime}\ (\text{Always}\ p))$
`lemma33`	$\vdash \forall p\ q.\ \text{Valid}\ ((\text{Always}\ p\ \text{And Sometime}\ q) \longrightarrow \text{Sometime}\ (p\ \text{And}\ q))$
`lemma34`	$\vdash \forall p.\ \text{Valid}\ (\text{Always}\ (p \longrightarrow q) \longrightarrow (\text{Always}\ p \longrightarrow \text{Always}\ q))$
`lemma35`	$\vdash \forall p.\ \text{Valid}\ (\text{Always}\ p \longrightarrow p)$
`lemma36`	$\vdash \forall p.\ \text{Next}\ (\text{Not}\ p) = \text{Not}\ (\text{Next}\ p)$
`lemma37`	$\vdash \forall p\ q.\ \text{Valid}\ (\text{Next}\ (p \longrightarrow q) \longrightarrow (\text{Next}\ p \longrightarrow \text{Next}\ q))$
`lemma38`	$\vdash \forall p.\ \text{Valid}\ (\text{Always}\ p \longrightarrow \text{Next}\ p)$
`lemma39`	$\vdash \forall p.\ \text{Valid}\ (\text{Always}\ p \longrightarrow \text{Next}\ (\text{Always}\ p))$
`lemma40`	$\vdash \forall p.\ \text{Valid}\ (p \longrightarrow \text{Sometime}\ p)$
`lemma41`	$\vdash \forall p\ q.\ \text{Valid}\ (\text{Always}\ (p \longrightarrow q) \longrightarrow (\text{Sometime}\ p \longrightarrow \text{Sometime}\ q))$
`lemma42`	$\vdash \forall p\ q.\ \text{Valid}\ (\text{Always}\ (p \longrightarrow q) \longrightarrow (\text{Next}\ p \longrightarrow \text{Next}\ q))$
`lemma43`	$\vdash \forall p\ q\ r.\ \text{Valid}\ (((\text{Always}\ p \longrightarrow \text{Always}\ q)\ \text{And}\ (\text{Always}\ q \longrightarrow \text{Always}\ r)) \longrightarrow (\text{Always}\ p \longrightarrow \text{Always}\ r))$
`lemma44`	$\vdash \forall p\ q\ r.\ \text{Valid}\ (((\text{Always}\ p \longrightarrow \text{Always}\ q)\ \text{And}\ (\text{Always}\ q \longrightarrow \text{Sometime}\ r)) \longrightarrow (\text{Always}\ p \longrightarrow \text{Sometime}\ r))$
`lemma45`	$\vdash \forall p\ q\ r.\ \text{Valid}\ (((\text{Always}\ p \longrightarrow \text{Always}\ q)\ \text{And}\ (\text{Always}\ q \longrightarrow \text{Next}\ r)) \longrightarrow (\text{Always}\ p \longrightarrow \text{Next}\ r))$
`lemma46`	$\vdash \forall p\ q\ r.\ \text{Valid}\ (((p \longrightarrow \text{Sometime}\ q)\ \text{And Always}\ (q \longrightarrow r)) \longrightarrow (p \longrightarrow \text{Sometime}\ r))$
`lemma47`	$\vdash \forall p.\ w\ 0\ \wedge\ \text{Valid}\ (w \longrightarrow \text{Next}\ w)\ \supset\ \text{Valid}\ (\text{Always}\ w)$
`lemma48`	$\vdash \forall p\ q.\ \text{Valid}\ (p \longrightarrow q)\ \supset\ \text{Valid}\ (\text{Sometime}\ p \longrightarrow \text{Sometime}\ q)$
`lemma49`	$\vdash \forall p\ q\ r.\ \text{Valid}\ (((p\ \text{And}\ q)\ \text{Until}\ r) \longrightarrow ((p\ \text{Until}\ r)\ \text{And}\ (q\ \text{Until}\ r)))$
`lemma50`	$\vdash \forall p\ q\ r.\ \text{Valid}\ ((\text{Always}\ p\ \text{And}\ (q\ \text{Until}\ r)) \longrightarrow ((p\ \text{And}\ q)\ \text{Until}\ (q\ \text{Until}\ r)))$
`lemma51`	$\vdash \forall p.\ \text{Valid}\ ((\text{Not}\ p\ \text{Until}\ p) \longrightarrow \text{Sometime}\ p)$

`lemma52`	$\vdash \forall p.\ \mathsf{Valid}\ p \supset (\forall n.\ (\exists t.\ \mathsf{Mapped}\ p\ n\ t))$
`lemma53`	$\vdash \forall p\ n.\ \mathsf{Valid}\ p \supset \mathsf{Mapped}\ p\ n\ (p\ \mathsf{Proj}\ n)$
`lemma54`	$\vdash \forall p\ n.\ \mathsf{Valid}\ p \supset p\ (p\ \mathsf{Proj}\ n)$
`lemma55`	$\vdash \forall p.\ \mathsf{Valid}\ p \supset (\forall n.\ (\exists\forall t.\ \mathsf{Mapped}\ p\ n\ t))$
`lemma56`	$\vdash \forall p\ n\ t\ t'.\ \mathsf{Valid}\ p \supset (\mathsf{Mapped}\ p\ n\ t \wedge \mathsf{Mapped}\ p\ n\ t' \supset (t = t'))$
`lemma57`	$\vdash \forall p.\ \mathsf{Valid}\ p \supset p\ \mathsf{Proj}\ n < p\ \mathsf{Proj}\ \mathsf{SUC}\ n$
`lemma58`	$\vdash \forall p.\ \mathsf{Valid}\ p \supset (\forall n\ t.\ p\ \mathsf{Proj}\ n < t \wedge t < p\ \mathsf{Proj}\ \mathsf{SUC}\ n \supset \neg p\ t)$
`lemma59`	$\vdash \forall p\ q.\ \mathsf{Valid}\ (p\ \mathsf{Equiv}\ q) \supset (\mathsf{Valid}\ p = \mathsf{Valid}\ q)$
`lemma60`	$\vdash \forall p\ q.\ \mathsf{Valid}\ (p \longrightarrow q) \supset (\mathsf{Valid}\ p \supset \mathsf{Valid}\ q)$
`lemma61`	$\vdash \forall p\ q.\ \mathsf{Valid}\ (p\ \mathsf{And}\ q) = \mathsf{Valid}\ p \wedge \mathsf{Valid}\ q$
`lemma62`	$\vdash \forall p.\ \mathsf{Valid}\ (p\ \mathsf{Equiv}\ p)$
`lemma63`	$\vdash \forall p.\ \mathsf{Valid}\ (p \longrightarrow p)$
`lemma64`	$\vdash \forall p\ q.\ \mathsf{Valid}\ (p\ \mathsf{Equiv}\ q) = \mathsf{Valid}\ (q\ \mathsf{Equiv}\ p)$
`lemma65`	$\vdash \forall p\ q.\ \mathsf{Not}\ (p\ \mathsf{And}\ q) = \mathsf{Not}\ p\ \mathsf{Or}\ \mathsf{Not}\ q$
`lemma66`	$\vdash \forall p\ q.\ \mathsf{Not}\ (p\ \mathsf{Or}\ q) = \mathsf{Not}\ p\ \mathsf{And}\ \mathsf{Not}\ q$
`lemma67`	$\vdash \forall p\ q.\ \mathsf{Not}\ p\ \mathsf{Or}\ q = p \longrightarrow q$
`lemma68`	$\vdash \forall p\ q\ r.\ \mathsf{Valid}\ (((p\ \mathsf{Equiv}\ q)\ \mathsf{And}\ (q\ \mathsf{Equiv}\ r)) \longrightarrow (p\ \mathsf{Equiv}\ r))$
`lemma69`	$\vdash \forall p\ q.\ (p \longrightarrow q)\ \mathsf{And}\ (q \longrightarrow p) = p\ \mathsf{Equiv}\ q$
`lemma70`	$\vdash \forall p\ q\ r.\ \mathsf{Valid}\ (((p \longrightarrow q)\ \mathsf{And}\ (q \longrightarrow r)) \longrightarrow (p \longrightarrow r))$
`lemma71`	$\vdash \forall p\ q\ r.\ \mathsf{Valid}\ (((p \longrightarrow q)\ \mathsf{And}\ (q\ \mathsf{Equiv}\ r)) \longrightarrow (p \longrightarrow r))$
`lemma72`	$\vdash \forall p\ q\ r.\ \mathsf{Valid}\ (((p\ \mathsf{Equiv}\ q)\ \mathsf{And}\ (q \longrightarrow r)) \longrightarrow (p \longrightarrow r))$
`lemma73`	$\vdash \forall p\ q\ r.\ \mathsf{Valid}\ (((p \longrightarrow q)\ \mathsf{And}\ (q \longrightarrow r)) \longrightarrow (p \longrightarrow (q\ \mathsf{And}\ r)))$
`lemma74`	$\vdash \forall p\ q\ r\ w.\ \mathsf{Valid}\ (((p\ \mathsf{Equiv}\ q)\ \mathsf{And}\ (r\ \mathsf{Equiv}\ w))$ $\longrightarrow ((p\ \mathsf{And}\ r)\ \mathsf{Equiv}\ (q\ \mathsf{And}\ w)))$
`lemma75`	$\vdash \forall p\ q\ r.\ \mathsf{Valid}\ (((p \longrightarrow (q \longrightarrow r))\ \mathsf{And}\ (p \longrightarrow q)) \longrightarrow (p \longrightarrow r))$
`lemma76`	$\vdash \forall p\ q\ r.\ p\ \mathsf{And}\ (q\ \mathsf{And}\ r) = (p\ \mathsf{And}\ q)\ \mathsf{And}\ r$

`lemma77`	$\vdash \forall p\ q\ r.\ p\ \mathsf{Or}\ (q\ \mathsf{Or}\ r) = (p\ \mathsf{Or}\ q)\ \mathsf{Or}\ r$
`lemma78`	$\vdash \forall p\ q\ r.\ p\ \mathsf{And}\ (q\ \mathsf{Or}\ r) = (p\ \mathsf{And}\ q)\ \mathsf{Or}\ (p\ \mathsf{And}\ r)$
`lemma79`	$\vdash \forall p\ q\ r.\ (p\ \mathsf{Or}\ q)\ \mathsf{And}\ r = (p\ \mathsf{And}\ r)\ \mathsf{Or}\ (q\ \mathsf{And}\ r)$
`lemma80`	$\vdash \forall p\ q\ r.\ p\ \mathsf{Or}\ (q\ \mathsf{And}\ r) = (p\ \mathsf{Or}\ q)\ \mathsf{And}\ (p\ \mathsf{Or}\ r)$
`lemma81`	$\vdash \forall p\ q\ r.\ (p\ \mathsf{And}\ q)\ \mathsf{Or}\ r = (p\ \mathsf{Or}\ r)\ \mathsf{And}\ (q\ \mathsf{Or}\ r)$
`lemma82`	$\vdash \forall p.\ p\ \mathsf{And}\ \mathsf{Not}\ p = \mathsf{False}$
`lemma83`	$\vdash \forall p.\ \mathsf{False}\ \mathsf{And}\ p = \mathsf{False}$
`lemma84`	$\vdash \forall p.\ p\ \mathsf{And}\ \mathsf{False} = \mathsf{False}$
`lemma85`	$\vdash \forall p.\ \mathsf{True}\ \mathsf{And}\ p = p$
`lemma86`	$\vdash \forall p.\ p\ \mathsf{And}\ \mathsf{True} = p$
`lemma87`	$\vdash \forall p.\ p\ \mathsf{Or}\ \mathsf{Not}\ p = \mathsf{True}$
`lemma88`	$\vdash \forall p.\ \mathsf{True}\ \mathsf{Or}\ p = \mathsf{True}$
`lemma89`	$\vdash \forall p.\ p\ \mathsf{Or}\ \mathsf{True} = \mathsf{True}$
`lemma90`	$\vdash \forall p.\ \mathsf{False}\ \mathsf{Or}\ p = p$
`lemma91`	$\vdash \forall p.\ p\ \mathsf{Or}\ \mathsf{False} = p$
`lemma92`	$\vdash \forall p\ q\ r\ w.\ (p\ \mathsf{And}\ q)\ \mathsf{And}\ (r\ \mathsf{And}\ w) = (p\ \mathsf{And}\ r)\ \mathsf{And}\ (q\ \mathsf{And}\ w)$
`lemma93`	$\vdash \forall p.\ p \longrightarrow \mathsf{False} = \mathsf{Not}\ p$
`lemma94`	$\vdash \forall p\ q.\ \mathsf{Valid}\ (p \longrightarrow q) \land \mathsf{Valid}\ p \supset \mathsf{Valid}\ q$
`lemma95`	$\vdash \forall p.\ \mathsf{Not}\ (\mathsf{Sometime}\ p) = \mathsf{Always}\ (\mathsf{Not}\ p)$
`lemma96`	$\vdash \forall p.\ \mathsf{Not}\ (\mathsf{Always}\ p) = \mathsf{Sometime}\ (\mathsf{Not}\ p)$
`lemma97`	$\vdash \forall p\ q.\ \mathsf{Valid}\ (\mathsf{Always}\ (p \longrightarrow q) \longrightarrow (\mathsf{Always}\ p \longrightarrow \mathsf{Always}\ q))$
`lemma98`	$\vdash \forall p.\ \mathsf{Valid}\ (\mathsf{Always}\ p \longrightarrow p)$
`lemma99`	$\vdash \forall p.\ \mathsf{Not}\ (\mathsf{Next}\ p) = \mathsf{Next}\ (\mathsf{Not}\ p)$
`lemma100`	$\vdash \forall p\ q.\ \mathsf{Valid}\ (\mathsf{Next}\ (p \longrightarrow q) \longrightarrow (\mathsf{Next}\ p \longrightarrow \mathsf{Next}\ q))$
`lemma101`	$\vdash \forall p.\ \mathsf{Valid}\ (\mathsf{Always}\ p \longrightarrow \mathsf{Next}\ p)$
`lemma102`	$\vdash \forall p.\ \mathsf{Valid}\ (\mathsf{Always}\ p \longrightarrow \mathsf{Next}\ (\mathsf{Always}\ p))$

`lemma103`	$\vdash \forall p.\, \mathsf{Valid}\ (\mathsf{Always}\ (p \longrightarrow \mathsf{Next}\ p) \longrightarrow (p \longrightarrow \mathsf{Always}\ p))$
`lemma104`	$\vdash \forall p\ q.\, p\ \mathsf{Until}\ q = q\ \mathsf{Or}\ (p\ \mathsf{And}\ \mathsf{Next}\ (p\ \mathsf{Until}\ q))$
`lemma105`	$\vdash \forall p\ q.\, \mathsf{Valid}\ ((p\ \mathsf{Until}\ q) \longrightarrow \mathsf{Sometime}\ q)$
`lemma106`	$\vdash \forall p\ q.\, p \longrightarrow q = \mathsf{Not}\ q \longrightarrow \mathsf{Not}\ p$
`lemma107`	$\vdash \forall p.\, \mathsf{Not}\ (\mathsf{Not}\ p) = p$
`lemma108`	$\vdash \forall p.\, \mathsf{Valid}\ p \supset \mathsf{Valid}\ (\mathsf{Always}\ p)$
`lemma109`	$\vdash \forall p.\, \mathsf{Valid}\ p \supset \mathsf{Valid}\ (\mathsf{Next}\ p)$
`lemma110`	$\vdash \forall p.\, \mathsf{Valid}\ (p \longrightarrow \mathsf{Sometime}\ p)$
`lemma111`	$\vdash \forall p.\, \mathsf{Valid}\ p \supset \mathsf{Valid}\ (\mathsf{Sometime}\ p)$
`lemma112`	$\vdash \forall p.\, \mathsf{Valid}\ (\mathsf{Next}\ p \longrightarrow \mathsf{Sometime}\ p)$
`lemma113`	$\vdash \forall p\ q.\, \mathsf{Valid}\ (p \longrightarrow q) \supset \mathsf{Valid}\ (\mathsf{Always}\ p \longrightarrow \mathsf{Always}\ q)$
`lemma114`	$\vdash \forall p\ q.\, \mathsf{Valid}\ (p \longrightarrow q) \supset \mathsf{Valid}\ (\mathsf{Sometime}\ p \longrightarrow \mathsf{Sometime}\ q)$
`lemma115`	$\vdash \forall p\ q.\, \mathsf{Valid}\ (p \longrightarrow q) \supset \mathsf{Valid}\ (\mathsf{Next}\ p \longrightarrow \mathsf{Next}\ q)$
`lemma116`	$\vdash \forall p.\, \mathsf{Valid}\ (p \longrightarrow \mathsf{Next}\ p) \supset \mathsf{Valid}\ (p \longrightarrow \mathsf{Always}\ p)$
`lemma117`	$\vdash \forall p.\, \mathsf{Always}\ p = \mathsf{Always}\ (\mathsf{Always}\ p)$
`lemma118`	$\vdash \forall p.\, \mathsf{Sometime}\ p = \mathsf{Sometime}\ (\mathsf{Sometime}\ p)$
`lemma119`	$\vdash \forall p\ q.\, \mathsf{Valid}\ (\mathsf{Always}\ (p \longrightarrow q)) \supset$ $\mathsf{Valid}\ (\mathsf{Sometime}\ p \longrightarrow \mathsf{Sometime}\ q)$
`lemma120`	$\vdash \forall p\ q.\, \mathsf{Always}\ (p\ \mathsf{And}\ q) = \mathsf{Always}\ p\ \mathsf{And}\ \mathsf{Always}\ q$
`lemma121`	$\vdash \forall p\ q.\, \mathsf{Sometime}\ (p\ \mathsf{Or}\ q) = \mathsf{Sometime}\ p\ \mathsf{Or}\ \mathsf{Sometime}\ q$
`lemma122`	$\vdash \forall p\ q.\, \mathsf{Valid}\ ((\mathsf{Always}\ p\ \mathsf{Or}\ \mathsf{Always}\ q) \longrightarrow \mathsf{Always}\ (p\ \mathsf{Or}\ q))$
`lemma123`	$\vdash \forall p\ q.\, \mathsf{Valid}\ (\mathsf{Sometime}\ (p\ \mathsf{And}\ q) \longrightarrow (\mathsf{Sometime}\ p\ \mathsf{And}\ \mathsf{Sometime}\ q))$
`lemma124`	$\vdash \forall p\ q.\, \mathsf{Valid}\ ((\mathsf{Always}\ p\ \mathsf{And}\ \mathsf{Sometime}\ q) \longrightarrow \mathsf{Sometime}\ (p\ \mathsf{And}\ q))$
`lemma125`	$\vdash \forall p\ q.\, \mathsf{Next}\ (p\ \mathsf{And}\ q) = \mathsf{Next}\ p\ \mathsf{And}\ \mathsf{Next}\ q$
`lemma126`	$\vdash \forall p\ q.\, \mathsf{Next}\ (p\ \mathsf{Or}\ q) = \mathsf{Next}\ p\ \mathsf{Or}\ \mathsf{Next}\ q$
`lemma127`	$\vdash \forall p\ q.\, \mathsf{Next}\ (p \longrightarrow q) = \mathsf{Next}\ p \longrightarrow \mathsf{Next}\ q$

`lemma128` $\vdash \forall p\ q.\ \mathsf{Next}\ (p\ \mathsf{Equiv}\ q) = \mathsf{Next}\ p\ \mathsf{Equiv}\ \mathsf{Next}\ q$

`lemma129` $\vdash \forall p.\ \mathsf{Valid}\ (p\ \mathsf{And}\ \mathsf{Sometime}\ (\mathsf{Not}\ p)) \supset$
$\mathsf{Valid}\ (\mathsf{Sometime}\ (p\ \mathsf{And}\ \mathsf{Next}\ p))$

`lemma130` $\vdash \forall p_1\ p_2\ q_1\ q_2.\ \mathsf{Valid}\ (p_1 \longrightarrow p_2)\ \wedge\ \mathsf{Valid}\ (p_2 \longrightarrow \mathsf{Always}\ q_1)\ \wedge$
$\mathsf{Valid}\ (q_1 \longrightarrow q_2)\ \supset\ \mathsf{Valid}\ (p_1 \longrightarrow \mathsf{Always}\ q_2)$

`lemma131` $\vdash \forall p_1\ p_2\ q_1\ q_2.\ \mathsf{Valid}\ (p_1 \longrightarrow p_2)\ \wedge\ \mathsf{Valid}\ (p_2 \longrightarrow \mathsf{Sometime}\ q_1)\ \wedge$
$\mathsf{Valid}\ (q_1 \longrightarrow q_2)\ \supset\ \mathsf{Valid}\ (p_1 \longrightarrow \mathsf{Sometime}\ q_2)$

`lemma132` $\vdash \forall p_1\ p_2\ q_1\ q_2.\ \mathsf{Valid}\ (p_1 \longrightarrow p_2)\ \wedge\ \mathsf{Valid}\ (p_2 \longrightarrow \mathsf{Next}\ q_1)\ \wedge$
$\mathsf{Valid}\ (q_1 \longrightarrow q_2)\ \supset\ \mathsf{Valid}\ (p_1 \longrightarrow \mathsf{Next}\ q_2)$

`lemma133` $\vdash \forall p\ q\ r.\ \mathsf{Valid}\ ((p \longrightarrow \mathsf{Always}\ q)\ \mathsf{And}\ (q \longrightarrow \mathsf{Always}\ r)) \supset$
$\mathsf{Valid}\ (p \longrightarrow \mathsf{Always}\ r)$

`lemma134` $\vdash \forall p\ q\ r.\ \mathsf{Valid}\ ((p \longrightarrow \mathsf{Sometime}\ q)\ \mathsf{And}\ (q \longrightarrow \mathsf{Sometime}\ r)) \supset$
$\mathsf{Valid}\ (p \longrightarrow \mathsf{Sometime}\ r)$

`lemma135` $\vdash \forall p\ q\ r.\ \mathsf{Valid}\ ((p \longrightarrow \mathsf{Sometime}\ q)\ \mathsf{And}$
$(r \longrightarrow (q\ \mathsf{And}\ (p\ \mathsf{And}\ \mathsf{Next}\ r)))) \supset \mathsf{Valid}\ (r \longrightarrow (p\ \mathsf{Until}\ q))$

`lemma136` $\vdash \forall p_1\ p_2\ q_1\ q_2.\ \mathsf{Valid}\ (p_1 \longrightarrow p_2)\ \wedge\ \mathsf{Valid}\ (q_1 \longrightarrow q_2) \supset$
$\mathsf{Valid}\ ((p_1\ \mathsf{Until}\ q_1) \longrightarrow (p_2\ \mathsf{Until}\ q_2))$

`lemma137` $\vdash \forall p\ q.\ \mathsf{Valid}\ ((\mathsf{Always}\ p\ \mathsf{And}\ \mathsf{Sometime}\ q) \longrightarrow (p\ \mathsf{Until}\ q))$

`lemma138` $\vdash \forall p\ q\ r.\ \mathsf{Valid}\ ((\mathsf{Always}\ p\ \mathsf{And}\ (q\ \mathsf{Until}\ r))$
$\longrightarrow ((p\ \mathsf{And}\ q)\ \mathsf{Until}\ (p\ \mathsf{And}\ r)))$

`lemma139` $\vdash \forall p\ q\ r.\ \mathsf{Valid}\ ((p\ \mathsf{Until}\ (q\ \mathsf{And}\ r)) \longrightarrow ((p\ \mathsf{Until}\ q)\ \mathsf{And}\ (p\ \mathsf{Until}\ r)))$

`lemma140` $\vdash \forall p\ q\ r.\ \mathsf{Valid}\ ((p\ \mathsf{Until}\ (q\ \mathsf{And}\ r)) \longrightarrow ((p\ \mathsf{Until}\ q)\ \mathsf{Until}\ r))$

B.3 Procedures of Proofs and Tactics

```
new_type_abbrev(`time`,":num");;

%<-----------------------------------------------------------

    General Tactics and Theorems

----------------------------------------------------------->%
```

```
let FORALL_EQ_TAC:tactic(asl,w)=
  (let t1,t2 = dest_eq w in
    let x1,p1 = dest_forall t1 and x2,p2 = dest_forall t2 in
      if (type_of x1)=(type_of x2) then
         (let x'=genvar(type_of x1) in
           [asl,mk_eq (p1,p2)],\[th]. FORALL_EQ (x') th)
       else fail ? failwith `FORALL_EQ_TAC`);;


letrec DEPTH_FIRST_CONV conv tm =
  FIRST_CONV
    [conv;
     RATOR_CONV (DEPTH_FIRST_CONV conv);
     RAND_CONV  (DEPTH_FIRST_CONV conv);
     ABS_CONV   (DEPTH_FIRST_CONV conv)]
    tm;;


let ONCE_LEFT_TO_RIGHT_DEPTH_FIRST_REWRITE_TAC =
  GEN_REWRITE_TAC DEPTH_FIRST_CONV basic_rewrites;;


letrec DEPTH_FST_CONV conv tm =
  FIRST_CONV
    [RATOR_CONV (DEPTH_FST_CONV conv);
     RAND_CONV  (DEPTH_FST_CONV conv);
     ABS_CONV   (DEPTH_FST_CONV conv);
     conv]
    tm;;


let ONCE_DEPTH_FST_REWRITE_TAC =
  GEN_REWRITE_TAC DEPTH_FST_CONV basic_rewrites;;


let change_var trm =
  let trm_s, trm_t = dest_var trm in
       mk_comb (mk_var(``^trm_s,":num->bool"),"(t:num)");;
```

```
let CONVERT thm =
  (let (bv,body) = strip_forall (concl thm) in
     let vars = map change_var bv in
       SPECL vars thm)
   ? failwith `CONVERT`;;


let LESSEQ = prove_thm (`LESSEQ`,
  "!a b. ~(a<b) ==> (b<=a)",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[IMP_DISJ_THM;LESS_CASES]);;


let SUC_LESS_SUC = prove_thm (`SUC_LESS_SUC`,
  "! t t'. (SUC t)<=t' ==> (SUC t)<=(SUC t')",
    REPEAT STRIP_TAC THEN REWRITE_TAC[LESS_EQ_MONO]
    THEN ASSUME_TAC (SPEC "t:num" LESS_EQ_SUC_REFL)
    THEN ASSUME_TAC LESS_EQ_TRANS THEN RES_TAC);


let NOT_EQ_REFL = prove_thm (`NOT_EQ_REFL`,
  "!a b. (~a = b) = (a = ~b)",
    REPEAT GEN_TAC THEN BOOL_CASES_TAC "a:bool"
    THEN REWRITE_TAC[]);;


let IMP_AND = prove_thm (`IMP_AND`,
  "!a b c. (a ==> b ==> c) = (a /\ b) ==> c",
    REPEAT GEN_TAC THEN BOOL_CASES_TAC "a:bool"
    THEN REWRITE_TAC[]);;


let LESS_EQ_TRANS = prove_thm (`LESS_EQ_TRANS`,
  "!a b c. (a<=b) /\ (b<=c) ==> (a<=c)",
    REPEAT GEN_TAC THEN REWRITE_TAC[LESS_OR_EQ]
    THEN STRIP_TAC THEN ASM_REWRITE_TAC[LESS_TRANS]
    THENL [MP_TAC (SPECL["a:num";"b:num";"c:num"] LESS_TRANS)
              THEN ASM_REWRITE_TAC[]
              THEN STRIP_TAC THEN ASM_REWRITE_TAC[];
```

```
          ASSUM_LIST (\t. (REWRITE_TAC[SYM (el 1 t)]))
          THEN ASM_REWRITE_TAC[]]);;

let IMP_CONTRAPOS = prove_thm (`IMP_CONTRAPOS`,
  "!a b. (a ==> b) = (~b ==> ~a)",
    REPEAT GEN_TAC THEN BOOL_CASES_TAC "a:bool"
    THEN REWRITE_TAC[]);;

let IMP_SELF = prove_thm (`IMP_SELF`,
  "!(a:num->bool) (b:num->bool). a n ==> (b n ==> a n)",
    REPEAT GEN_TAC THEN BOOL_CASES_TAC "(a:num->bool) n"
    THEN REWRITE_TAC[]);;

%<-----------------------------------------------------------
     Temporal Definitions
----------------------------------------------------------->%

let And = new_infix_definition (`And_DEF`,
  " And x y = \(t:time). x t /\ y t");;

let Or = new_infix_definition (`Or_DEF`,
  "Or x y = \(t:time). x t \/ y t");;

let Not = new_definition (`Not_DEF`,
  "Not x = \(t:time). ~(x t)");;

let Imp = new_infix_definition (`Imp_DEF`,
  "$--> x y = \(t:time) . (x t) ==> (y t)");;

let Xor2 = new_definition(`Xor2_DEF`,
  "Xor2 (x,y) =\(t:time). (x t \/ y t) /\ ~(x t /\ y t)");;

let Xor3 = new_definition(`Xor3_DEF`,
  "Xor3 (x,y,z) =\(t:time). (x t \/ y t \/ z t)
       /\ ~((x t /\ y t) \/ (y t /\ z t) \/ (z t /\ x t))");;
```

```
let True = new_definition (`True_DEF`, "True = \t. T");;


let False = new_definition (`False_DEF`, "False = \t. F");;


let Next = new_definition (`Next_DEF`,
  "Next p = \t. ((p (SUC t)):bool)");;


let Until = new_infix_definition (`Until_DEF`,
  "Until p q = \t. (?t1. (t<=t1) /\ (q t1) /\
         (!t2. ((t<=t2) /\ (t2<t1)) ==> (p t2)))");;


let Sometime = new_definition (`Sometime_DEF`,
  "Sometime (p:time->bool) = \t. ?t'. (t<=t') /\ p t'");;


let Always = new_definition (`Always_DEF`,
  "Always (p:time->bool) = \t. !t'. (t<=t') ==> p t'");;


let Eq = new_infix_definition (`Eq_DEF`,
  "Eq (f:time->*) c = \t. f t = c");;


let Equiv = new_infix_definition (`Equiv_DEF`,
  "Equiv (p:time->bool) (q:time->bool) =
        \t. (p t) = (q t)");;


let First = new_definition (`First_DEF`,
  "First p t = p t /\ (!t'. t'<t ==> ~(p t'))");;


let Last = new_definition (`Last_DEF`,
  "Last p (t1,t2) =
      (!t. t1<t /\ t<t2 ==> ~(p t)) /\ (p t2)");;


let Fin = new_definition (`Fin_DEF`,
  "Fin (p:time->bool) (t1:time,t2) = (t1<t2) /\ p t2");;
```

```
let Keep = new_definition ('Keep_DEF',
  "Keep p (t1,t2) =
     (t1<t2) /\ (!t. t1<t /\ t<t2 ==> (p t))");;

let Length = new_prim_rec_definition ('Length_DEF',
  "(Length p 0 = \t. p t) /\
   (Length p (SUC n) = \t. Length p n t /\ ~p(SUC (n+t)))");;

let Len = new_definition ('Len_DEF',
  "Len p n = \t'. Length p n (t'-n)");;

let Mapped = new_prim_rec_definition ('Mapped_DEF',
  "(Mapped p 0 = \t. First p t) /\
   (Mapped p (SUC n) =
        \t.?t'. Mapped p n t' /\ Last p (t',t))");;

let Proj = new_infix_definition ('Proj_DEF',
  "Proj p n = @t. Mapped p n t");;

let Valid = new_definition ('Valid_DEF',
  "Valid p = !t. p t");;

let When = new_infix_definition ('When_DEF',
  "When p b = !n. b(p Proj n)");;

let Upon = new_infix_definition ('Upon_DEF',
  "Upon p b = \p'. \t. p When b => p' t | p t");;

let Timeout = new_prim_rec_definition ('Timeout_DEF',
  "(Timeout 0 b = T) /\
   (Timeout (SUC t) b = b => F | Timeout t b)");;

let Within = new_infix_definition ('Within_DEF',
  "Within b p = \t. ~(p t /\ Sometime b t)");;


%<---------------------------------------------------------

     Temporal Theorems

--------------------------------------------------------->%

let lemma1 = prove_thm ('lemma1',
  "!x. x And x = x",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[And] THEN BETA_TAC
    THEN GEN_TAC THEN REFL_TAC);;


let lemma2 = prove_thm ('lemma2',
  "!x y. x And y = y And x",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[And] THEN BETA_TAC
    THEN ONCE_DEPTH_FST_REWRITE_TAC[CONJ_SYM]
    THEN GEN_TAC THEN REFL_TAC);;


let lemma3 = prove_thm ('lemma3',
  "!x y z. (x And y) And z = x And (y And z)",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[And] THEN BETA_TAC
    THEN GEN_TAC THEN BOOL_CASES_TAC "(x:time->bool) n"
    THEN REWRITE_TAC[]);;


let lemma4 = prove_thm ('lemma4',
  "!x. x And True = x",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[And;True]
    THEN BETA_TAC THEN GEN_TAC THEN REFL_TAC);;


let lemma5 = prove_thm ('lemma5',
  "!x. x And False = False",
    REWRITE_TAC[And;False]);;
```

```
let lemma6 = prove_thm (`lemma6`,
  "!x. x Or x = x",
    GEN_TAC THEN REWRITE_TAC[Or]
    THEN CONV_TAC FUN_EQ_CONV
    THEN BETA_TAC THEN GEN_TAC
    THEN REFL_TAC);;

let lemma7 = prove_thm (`lemma7`,
  "!x y. x Or y = y Or x",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Or] THEN BETA_TAC THEN GEN_TAC
    THEN BOOL_CASES_TAC "(x:time->bool) n"
    THEN REWRITE_TAC[]);;

let lemma8 = prove_thm (`lemma8`,
  "!x y z. (x Or y) Or z = x Or (y Or z)",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Or] THEN BETA_TAC
    THEN GEN_TAC
    THEN BOOL_CASES_TAC "(x:time->bool) n"
    THEN REWRITE_TAC[]);;

let lemma9 = prove_thm (`lemma9`,
  "!x. x Or True = True",
    REWRITE_TAC[Or;True]);;

let lemma10 = prove_thm (`lemma10`,
  "!x. x Or False = x",
    REWRITE_TAC[Or;False] THEN GEN_TAC
    THEN CONV_TAC FUN_EQ_CONV THEN BETA_TAC
    THEN GEN_TAC THEN REFL_TAC);;

let lemma11 = prove_thm (`lemma11`,
  "!x. Xor2(x,x) = False",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Xor2;False]
    THEN BETA_TAC THEN GEN_TAC
    THEN BOOL_CASES_TAC "(x:time->bool) n"
    THEN REWRITE_TAC[]);;

let lemma12 = prove_thm (`lemma12`,
  "!x y. Xor2(x,y) = Xor2(y,x)",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Xor2] THEN BETA_TAC
    THEN GEN_TAC
    THEN BOOL_CASES_TAC "(x:time->bool) n"
    THEN REWRITE_TAC[]);;

let lemma13 = prove_thm (`lemma13`,
  "!x. Xor2(x,True) = Not x",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Xor2;True;Not]
    THEN BETA_TAC);;

let lemma14 = prove_thm (`lemma14`,
  "!x. Xor2(x,y) = Xor3(x,y,False)",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Xor2;Xor3;False]
    THEN BETA_TAC);;

let lemma15 = prove_thm (`lemma15`,
  "(Not True = False) /\ (Not False = True)",
    REWRITE_TAC[True;False;Not]);;

let lemma16 = prove_thm (`lemma16`,
  "!p t1 t2. (t1<t2) /\ Last p (t1,t2) =
        Keep (Not p)(t1,t2) /\ Fin p(t1,t2)",
    REPEAT GEN_TAC THEN REWRITE_TAC[Fin;Last;Keep;Not]
    THEN BETA_TAC THEN EQ_TAC THEN REPEAT STRIP_TAC
    THEN RES_TAC THEN ASM_REWRITE_TAC[]);;

let lemma17 = prove_thm ('lemma17',
  "!p. Sometime p = True Until p",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Sometime;True;Until]);;

let lemma18 = prove_thm ('lemma18',
  "!p. Always p = Not(Sometime (Not p))",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Always;Sometime;Not]
    THEN BETA_TAC
    THEN CONV_TAC (DEPTH_CONV NOT_EXISTS_CONV)
    THEN REWRITE_TAC[DE_MORGAN_THM;IMP_DISJ_THM]);;

let lemma19 = prove_thm ('lemma19',
  "!p. Valid (p --> Sometime p)",
    GEN_TAC THEN REWRITE_TAC[Valid;Sometime;Imp]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "t"
    THEN ASM_REWRITE_TAC[LESS_EQ_REFL]);;

let lemma20 = prove_thm ('lemma20',
  "!p. Valid (Next p --> Sometime p)",
    GEN_TAC
    THEN REWRITE_TAC[Valid;Imp;Next;Sometime]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "SUC t"
    THEN ASM_REWRITE_TAC[LESS_EQ_SUC_REFL]);;

let lemma21 = prove_thm ('lemma21',
  "!p. Sometime (Not p) = Not (Always p)",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Sometime;Not;Always]
    THEN BETA_TAC
    THEN CONV_TAC (DEPTH_CONV NOT_FORALL_CONV)
    THEN REWRITE_TAC[IMP_DISJ_THM;DE_MORGAN_THM]);;

let lemma22 = prove_thm (`lemma22`,
  "!p q. Valid (Always(p-->q) -->
        (Sometime p --> Sometime q))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Always;Imp;Sometime]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN RES_TAC THEN EXISTS_TAC "t'"
    THEN ASM_REWRITE_TAC[]);;

let lemma23 = prove_thm (`lemma23`,
  "!p q. Always (p And q) = (Always p) And (Always q)",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Always;And] THEN BETA_TAC
    THEN GEN_TAC THEN EQ_TAC
    THEN REPEAT STRIP_TAC THEN RES_TAC);;

let lemma24 = prove_thm (`lemma24`,
  "!p q. Sometime (p Or q) = (Sometime p) Or (Sometime q)",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Sometime;Or]
    THEN BETA_TAC THEN GEN_TAC THEN EQ_TAC
    THENL [REPEAT STRIP_TAC
           THEN CONV_TAC OR_EXISTS_CONV
           THEN EXISTS_TAC "t'"
           THEN ASM_REWRITE_TAC[];
           CONV_TAC (DEPTH_CONV OR_EXISTS_CONV)
           THEN CONV_TAC LEFT_IMP_EXISTS_CONV
           THEN REPEAT STRIP_TAC
```

```
          THEN EXISTS_TAC "t'"
          THEN ASM_REWRITE_TAC[]]);;

let lemma25 = prove_thm (`lemma25`,
  "!p q. Valid ((Always p And Always q) --> Always(p Or q))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Imp;Always;And;Or]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN RES_TAC THEN ASM_REWRITE_TAC[]);;

let lemma26 = prove_thm (`lemma26`,
  "!p q. Valid (Sometime (p And q)
          --> (Sometime p) And (Sometime q))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Sometime;And;Imp]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "t'" THEN ASM_REWRITE_TAC[]);;

let lemma27 = prove_thm (`lemma27`,
  "!p q. Next (p And q) = (Next p) And (Next q)",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Next;And] THEN BETA_TAC
    THEN GEN_TAC THEN REFL_TAC);;

let lemma28 = prove_thm (`lemma28`,
  "!p q. Next (p Or q) = (Next p) Or (Next q)",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Next;Or]
    THEN BETA_TAC THEN GEN_TAC THEN REFL_TAC);;

let lemma29 = prove_thm (`lemma29`,
  "!p q. Next (p-->q) = Next p --> Next q",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Next;Imp]
    THEN BETA_TAC THEN GEN_TAC THEN REFL_TAC);;


let lemma30 = prove_thm (`lemma30`,
  "!p q. Valid ((Always p And Sometime q)-->(p Until q))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Always;Imp;And;Sometime;Until]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "t'" THEN ASM_REWRITE_TAC[]
    THEN REPEAT STRIP_TAC THEN RES_TAC);;


let lemma31 = prove_thm (`lemma31`,
  "!p. Valid (Sometime(Next p) --> Next(Sometime p))",
    GEN_TAC
    THEN REWRITE_TAC[Sometime;Valid;Next;Imp]
    THEN BETA_TAC THEN GEN_TAC
    THEN CONV_TAC (DEPTH_CONV LEFT_IMP_EXISTS_CONV)
    THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "SUC t'"
    THEN ASM_REWRITE_TAC[LESS_EQ_MONO]);;


let lemma32 = prove_thm (`lemma32`,
  "!p. Valid (Always(Sometime (Always p))
       --> Sometime(Always p))",
    GEN_TAC
    THEN REWRITE_TAC [Always;Sometime;Imp;Valid]
    THEN BETA_TAC THEN GEN_TAC
    THEN CONV_TAC LEFT_IMP_FORALL_CONV
    THEN EXISTS_TAC "t"
    THEN REWRITE_TAC[LESS_EQ_REFL]);;


let lemma33 = prove_thm (`lemma33`,
  "!p q. Valid ((Always p And Sometime q)
       --> Sometime (p And q))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Imp;And;Sometime;Always]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "t'" THEN RES_TAC
    THEN ASM_REWRITE_TAC[]);;

let lemma34 = prove_thm ('lemma34',
  "!p. Valid (Always(p-->q) --> Always p --> Always q)",
    GEN_TAC THEN REWRITE_TAC[Valid;Always;Imp]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN RES_TAC);;

let lemma35 = prove_thm ('lemma35',
  "!p. Valid ((Always p)-->p)",
    GEN_TAC THEN REWRITE_TAC[Valid;Always;Imp]
    THEN BETA_TAC
    THEN CONV_TAC (DEPTH_CONV LEFT_IMP_FORALL_CONV)
    THEN GEN_TAC THEN EXISTS_TAC "t"
    THEN REWRITE_TAC[LESS_EQ_REFL]);;

let lemma36 = prove_thm ('lemma36',
  "!p. Next (Not p) = Not (Next p)",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Next;Not] THEN BETA_TAC
    THEN GEN_TAC THEN REFL_TAC);;

let lemma37 = prove_thm ('lemma37',
  "!p q. Valid (Next(p-->q) --> (Next p --> Next q))",
    REPEAT GEN_TAC THEN REWRITE_TAC[Valid;Next;Imp]
    THEN BETA_TAC THEN GEN_TAC THEN STRIP_TAC);;

let lemma38 = prove_thm ('lemma38',
  "!p. Valid ((Always p)-->Next p)",
    GEN_TAC
    THEN REWRITE_TAC[Valid;Always;Next;Imp]
    THEN BETA_TAC
    THEN CONV_TAC (DEPTH_CONV LEFT_IMP_FORALL_CONV)
    THEN GEN_TAC THEN EXISTS_TAC "SUC t"
    THEN REWRITE_TAC[LESS_EQ_SUC_REFL]);;

let lemma39 = prove_thm (`lemma39`,
  "!p. Valid (Always p --> Next (Always p))",
    GEN_TAC
    THEN REWRITE_TAC[Valid;Always;Imp;Next]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN ASSUME_TAC (SPEC "t" LESS_EQ_SUC_REFL)
    THEN ASSUME_TAC LESS_EQ_TRANS
    THEN RES_TAC THEN RES_TAC);;

let lemma40 = prove_thm (`lemma40`,
  "!p. Valid (p --> Sometime p)",
    GEN_TAC
    THEN REWRITE_TAC[Valid;Imp;Sometime]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "t"
    THEN ASM_REWRITE_TAC[LESS_EQ_REFL]);;

let lemma41 = prove_thm (`lemma41`,
  "!p q. Valid (Always(p-->q)
       --> (Sometime p --> Sometime q))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Always;Imp;Sometime]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "t'" THEN ASM_REWRITE_TAC[]
    THEN RES_TAC);;

let lemma42 = prove_thm (`lemma42`,
  "!p q. Valid (Always(p-->q) --> (Next p --> Next q))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Always;Imp;Next]
    THEN BETA_TAC
    THEN CONV_TAC(DEPTH_CONV LEFT_IMP_FORALL_CONV)
    THEN GEN_TAC THEN EXISTS_TAC "SUC t"
    THEN REWRITE_TAC[LESS_EQ_SUC_REFL]);;

let lemma43 = prove_thm (`lemma43`,
  "!p q r.
     Valid (((Always p --> Always q) And
             (Always q --> Always r))
          --> (Always p --> Always r))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Always;Imp;And]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN ASSUM_LIST (\th.
       ASSUME_TAC(REWRITE_RULE[(el 2 th)] (el 4 th)))
    THEN ASSUM_LIST (\th.
       ASSUME_TAC(REWRITE_RULE[(el 1 th)] (el 4 th)))
    THEN RES_TAC);;

let lemma44 = prove_thm (`lemma44`,
  "!p q r.
      Valid (((Always p --> Always q) And
              (Always q --> Sometime r))
            --> (Always p --> Sometime r))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Always;Imp;And;Sometime]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN ASSUM_LIST (\th.
       ASSUME_TAC(REWRITE_RULE[(el 1 th)] (el 3 th)))
    THEN ASSUM_LIST (\th.
    ASM_REWRITE_TAC[REWRITE_RULE[(el 1 th)] (el 3 th)]));;

let lemma45 = prove_thm (`lemma45`,
  "!p q r. Valid(
          ((Always p --> Always q) And (Always q --> Next r))
                   --> (Always p --> Next r))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Always;Imp;And;Next;Valid]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN ASSUM_LIST (\th. ASSUME_TAC(
         REWRITE_RULE[(el 1 th)] (el 3 th)))
    THEN ASSUM_LIST (\th. ASM_REWRITE_TAC[
         REWRITE_RULE[(el 1 th)] (el 3 th)]));;

let lemma46 = prove_thm ('lemma46',
  "!p q r. Valid(
        ((p-->Sometime q) And Always (q-->r))
                 --> (p-->Sometime r))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Sometime;And;Imp;Always]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN ASSUM_LIST (\th. ASSUME_TAC(
         REWRITE_RULE[(el 1 th)] (el 3 th)))
    THEN POP_ASSUM MP_TAC
    THEN CONV_TAC LEFT_IMP_EXISTS_CONV
    THEN REPEAT STRIP_TAC THEN EXISTS_TAC "t'"
    THEN RES_TAC THEN ASM_REWRITE_TAC[]);;

let lemma47 = prove_thm ('lemma47',
  "!p. w 0 /\ Valid(w-->Next w) ==> Valid(Always w)",
    GEN_TAC
    THEN REWRITE_TAC[Imp;Next;Valid;Always]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN ASSUME_TAC (SPEC "w:num->bool" INDUCTION)
    THEN RES_TAC THEN ASM_REWRITE_TAC[]);;

let lemma48 = prove_thm ('lemma48',
  "!p q. Valid(p-->q) ==> Valid(Sometime p --> Sometime q)",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Imp;Sometime]
    THEN BETA_TAC THEN DISCH_TAC
    THEN REPEAT STRIP_TAC THEN EXISTS_TAC "t'"
    THEN RES_TAC THEN ASM_REWRITE_TAC[]);;

let lemma49 = prove_thm (`lemma49`,
  "!p q r. Valid(((p And q) Until r)
            --> ((p Until r)And(q Until r)))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;And;Until;Imp]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "t1" THEN ASM_REWRITE_TAC[]
    THEN GEN_TAC THEN POP_ASSUM MP_TAC
    THEN CONV_TAC(DEPTH_CONV LEFT_IMP_FORALL_CONV)
    THEN EXISTS_TAC "t2" THEN REPEAT STRIP_TAC
    THEN RES_TAC);;

let lemma50 = prove_thm (`lemma50`,
  "!p q r. Valid(((Always p)And(q Until r))
              -->((p And q)Until(q Until r)))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Always;And;Until;Imp]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "t1" THEN ASM_REWRITE_TAC[]
    THEN REPEAT STRIP_TAC
    THENL [EXISTS_TAC "t1"
           THEN ASM_REWRITE_TAC[LESS_EQ_REFL]
           THEN GEN_TAC THEN POP_ASSUM MP_TAC
           THEN CONV_TAC (DEPTH_CONV LEFT_IMP_FORALL_CONV)
           THEN EXISTS_TAC "t2" THEN REPEAT STRIP_TAC
           THEN ASSUME_TAC(
             SPECL ["t:num";"t1:num";"t2:num"] LESS_EQ_TRANS)
           THEN RES_TAC;RES_TAC;RES_TAC]);

let lemma51 = prove_thm ('lemma51',
  "!p. Valid (((Not p) Until p) --> Sometime p)",
    GEN_TAC
    THEN REWRITE_TAC[Valid;Not;Until;Imp;Sometime]
    THEN BETA_TAC THEN GEN_TAC
    THEN CONV_TAC(DEPTH_CONV LEFT_IMP_EXISTS_CONV)
    THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "t1"
    THEN ASM_REWRITE_TAC[]);;

let lemma52 = prove_thm ('lemma52',
  "!p. Valid p ==> !n. ?t. Mapped p n t",
        REWRITE_TAC [Valid]
        THEN GEN_TAC THEN DISCH_TAC THEN INDUCT_TAC
        THENL [REWRITE_TAC[Mapped;First]
               THEN BETA_TAC THEN EXISTS_TAC "0"
               THEN REWRITE_TAC[NOT_LESS_0]
               THEN POP_ASSUM  MP_TAC
               THEN REPEAT STRIP_TAC
               THEN ASM_REWRITE_TAC[];
               POP_ASSUM MP_TAC
               THEN REWRITE_TAC[Mapped]
               THEN BETA_TAC THEN REPEAT STRIP_TAC
               THEN EXISTS_TAC "0" THEN EXISTS_TAC "t"
               THEN ASM_REWRITE_TAC[]
               THEN REWRITE_TAC[Last;NOT_LESS_0]
               THEN ASM_REWRITE_TAC[]]);;

let lemma53 = prove_thm ('lemma53',
  "!p n. Valid p ==> Mapped p n (p Proj n)",
        REWRITE_TAC[Valid;Proj]
        THEN CONV_TAC (DEPTH_CONV SELECT_CONV)
        THEN GEN_TAC
        THEN INDUCT_TAC
        THENL [REWRITE_TAC[Mapped;First]
               THEN BETA_TAC
               THEN DISCH_TAC
               THEN EXISTS_TAC "0"
               THEN ASM_REWRITE_TAC[NOT_LESS_0];
               STRIP_TAC THEN RES_TAC
               THEN EXISTS_TAC "t"
               THEN REWRITE_TAC[Mapped]
               THEN BETA_TAC
               THEN EXISTS_TAC "t"
               THEN ASM_REWRITE_TAC[Last;LESS_ANTISYM]]
    );;

let lemma54 = prove_thm (`lemma54`,
  "!p n. Valid p ==> p (Proj p n)",
        GEN_TAC THEN REWRITE_TAC[Valid]
        THEN INDUCT_TAC
        THENL [REWRITE_TAC[Proj;Mapped]
               THEN BETA_TAC THEN DISCH_TAC
               THEN ASM_REWRITE_TAC[First];
               REPEAT STRIP_TAC
               THEN REWRITE_TAC[Proj;Mapped]
               THEN BETA_TAC THEN ASM_REWRITE_TAC[]]);;

let lemma55 = prove_thm (`lemma55`,
  "!p. Valid p ==> !n. ?!t. Mapped p n t",
        REWRITE_TAC[Valid] THEN GEN_TAC THEN DISCH_TAC
        THEN INDUCT_TAC
        THENL [CONV_TAC EXISTS_UNIQUE_CONV
               THEN REWRITE_TAC[Mapped;First]
               THEN BETA_TAC THEN STRIP_TAC
               THENL [EXISTS_TAC "0"
```

```
                    THEN ASM_REWRITE_TAC[NOT_LESS_0];
                    ASM_REWRITE_TAC[]
                    THEN REPEAT STRIP_TAC
                    THEN POP_ASSUM MP_TAC
                    THEN CONV_TAC LEFT_IMP_FORALL_CONV
                    THEN EXISTS_TAC "t"
                    THEN POP_ASSUM MP_TAC
                    THEN CONV_TAC LEFT_IMP_FORALL_CONV
                    THEN EXISTS_TAC "t'"
                    THEN REPEAT STRIP_TAC
                    THEN ASSUME_TAC LESSEQ
                    THEN RES_TAC
                    THEN ASSUME_TAC LESS_EQUAL_ANTISYM
                    THEN RES_TAC];
              POP_ASSUM MP_TAC
              THEN CONV_TAC (DEPTH_CONV EXISTS_UNIQUE_CONV)
              THEN REPEAT STRIP_TAC
              THENL [EXISTS_TAC "0"
                    THEN REWRITE_TAC[Mapped]
                    THEN BETA_TAC THEN EXISTS_TAC "t"
                    THEN ASM_REWRITE_TAC[Last;NOT_LESS_0];
                    POP_ASSUM MP_TAC THEN POP_ASSUM MP_TAC
                    THEN REWRITE_TAC[Mapped]
                    THEN BETA_TAC THEN REPEAT STRIP_TAC
                    THEN RES_TAC]]);;


let lemma56 = prove_thm ('lemma56',
  "!p n t t'. Valid p ==>
          ((Mapped p n t /\ Mapped p n t') ==> (t=t'))",
     GEN_TAC THEN REWRITE_TAC[Valid]
     THEN INDUCT_TAC
     THENL [REWRITE_TAC[Mapped;First]
           THEN BETA_TAC THEN REPEAT STRIP_TAC
           THEN POP_ASSUM MP_TAC
```

```
            THEN CONV_TAC LEFT_IMP_FORALL_CONV
            THEN EXISTS_TAC "t" THEN ASM_REWRITE_TAC[]
            THEN POP_ASSUM(\th1.POP_ASSUM(\th2.
                 (ASSUME_TAC th1 THEN (ASSUME_TAC th2))))
            THEN POP_ASSUM MP_TAC
            THEN CONV_TAC(DEPTH_CONV LEFT_IMP_FORALL_CONV)
            THEN EXISTS_TAC "t'" THEN ASM_REWRITE_TAC[]
            THEN REPEAT STRIP_TAC THEN ASSUME_TAC LESSEQ
            THEN RES_TAC THEN ASSUME_TAC LESS_EQUAL_ANTISYM
            THEN RES_TAC;
            POP_ASSUM MP_TAC THEN REWRITE_TAC[Mapped]
            THEN BETA_TAC THEN REPEAT STRIP_TAC
            THEN RES_TAC THEN RES_TAC]);;


let lemma57 = prove_thm (`lemma57`,
  "!p. Valid p ==> (p Proj n) < (p Proj (SUC n))",
        GEN_TAC THEN REWRITE_TAC[Valid]
        THEN CONV_TAC (DEPTH_CONV LEFT_IMP_FORALL_CONV)
        THEN STRIP_TAC THEN POP_ASSUM (\th. ALL_TAC)
        THEN POP_ASSUM MP_TAC
        THEN CONV_TAC (DEPTH_CONV LEFT_IMP_FORALL_CONV)
        THEN EXISTS_TAC "n"
        THEN CONV_TAC (DEPTH_CONV LEFT_IMP_FORALL_CONV)
        THEN EXISTS_TAC "SUC n"
        THEN ASM_REWRITE_TAC[LESS_SUC_REFL]);;


let lemma58 = prove_thm (`lemma58`,
  "!p. Valid p ==>
     !n t. ((p Proj n)<t /\ t<(p Proj (SUC n))) ==> ~p t",
        GEN_TAC THEN REWRITE_TAC[Valid]
        THEN REPEAT STRIP_TAC THEN RES_TAC);;


%<-----------------------------------------------------------

     More Temporal Theorems
```

```
------------------------------------------------------------>%

let lemma59 = prove_thm ('lemma59',
  "!p q. Valid (p Equiv q) ==> (Valid p = Valid q)",
    REPEAT GEN_TAC THEN REWRITE_TAC[Valid;Equiv]
    THEN BETA_TAC THEN STRIP_TAC
    THEN ASM_REWRITE_TAC[]);;

let lemma60 = prove_thm ('lemma60',
  "!p q. Valid (p-->q) ==> Valid p ==> Valid q",
    REPEAT GEN_TAC THEN REWRITE_TAC[Valid;Imp]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN ASSUM_LIST (\t. MATCH_MP_TAC (el 2 t))
    THEN ASM_REWRITE_TAC[]);;

let lemma61 = prove_thm ('lemma61',
  "!p q. Valid (p And q) = Valid p /\ Valid q",
    REPEAT GEN_TAC THEN REWRITE_TAC[Valid;And]
    THEN BETA_TAC THEN CONV_TAC (DEPTH_CONV AND_FORALL_CONV)
    THEN REFL_TAC);;

let lemma62 = prove_thm ('lemma62',
  "!p. Valid (p Equiv p)",
    GEN_TAC THEN REWRITE_TAC[Valid;Equiv]
    THEN BETA_TAC);;

let lemma63 = prove_thm ('lemma63',
  "!p. Valid (p --> p)",
    GEN_TAC THEN REWRITE_TAC[Valid;Imp]
    THEN BETA_TAC);;

let lemma64 = prove_thm ('lemma64',
  "!p q. Valid (p Equiv q) = Valid (q Equiv p)",
    REPEAT GEN_TAC THEN REWRITE_TAC[Valid;Equiv]
    THEN BETA_TAC THEN EQ_TAC THEN STRIP_TAC
```

```
   THEN ASM_REWRITE_TAC[]);;

let lemma65 = prove_thm (`lemma65`,
  "!p q. Not(p And q) = (Not p) Or (Not q)",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Not;And;Or] THEN BETA_TAC
    THEN REWRITE_TAC[DE_MORGAN_THM]
    THEN REFL_TAC);;

let lemma66 = prove_thm (`lemma66`,
  "!p q. Not(p Or q) = (Not p) And (Not q)",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Not;Or;And]
    THEN BETA_TAC THEN REWRITE_TAC[DE_MORGAN_THM]
    THEN REFL_TAC);;

let lemma67 = prove_thm (`lemma67`,
  "!p q. (Not p) Or q = (p-->q)",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Not;Or;Imp] THEN BETA_TAC
    THEN REWRITE_TAC[IMP_DISJ_THM]);;

let lemma68 = prove_thm (`lemma68`,
  "!p q r. Valid (((p Equiv q) And (q Equiv r))
        --> (p Equiv r))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Equiv;And;Imp]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[]);;

let lemma69 = prove_thm (`lemma69`,
  "!p q. ((p-->q) And (q-->p)) = (p Equiv q)",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Imp;And;Equiv]
    THEN BETA_TAC THEN GEN_TAC THEN EQ_TAC
    THENL [STRIP_TAC THEN EQ_TAC
           THEN ASM_REWRITE_TAC[];
           STRIP_TAC THEN ASM_REWRITE_TAC[]]);;

let lemma70 = prove_thm (`lemma70`,
  "!p q r. Valid (((p-->q) And (q-->r)) --> (p-->r))",
    REPEAT GEN_TAC THEN REWRITE_TAC[Imp;And;Valid]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN REPEAT RES_TAC);;

let lemma71 = prove_thm (`lemma71`,
  "!p q r. Valid (((p-->q) And (q Equiv r)) --> (p-->r))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Imp;And;Equiv]
    THEN BETA_TAC THEN REPEAT STRIP_TAC THEN RES_TAC
    THEN ASSUM_LIST (\th. ASM_REWRITE_TAC[SYM (el 3 th)]));;

let lemma72 = prove_thm (`lemma72`,
  "!p q r. Valid (((p Equiv q) And (q-->r)) --> (p-->r))",
    REPEAT GEN_TAC THEN REWRITE_TAC[Valid;Imp;And;Equiv]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN REPEAT RES_TAC);;

let lemma73 = prove_thm (`lemma73`,
  "!p q r. Valid (((p-->q) And (q-->r))
          --> (p-->(q And r)))",
    REPEAT GEN_TAC THEN REWRITE_TAC[Valid;And;Imp]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN RES_TAC THEN RES_TAC);;

let lemma74 = prove_thm (`lemma74`,
  "!p q r w. Valid (((p Equiv q) And (r Equiv w)) -->
                    ((p And r) Equiv (q And w)))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Equiv;And;Imp]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[]);;

let lemma75 = prove_thm (`lemma75`,
  "!p q r. Valid (((p-->(q-->r)) And (p-->q)) --> (p-->r))",
    REPEAT GEN_TAC THEN REWRITE_TAC[Valid;Imp;And]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN RES_TAC);;

let lemma76 = prove_thm (`lemma76`,
  "!p q r. p And (q And r) = (p And q) And r",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[And] THEN BETA_TAC
    THEN GEN_TAC THEN BOOL_CASES_TAC "(q:time->bool) n"
    THEN REWRITE_TAC[]);;

let lemma77 = prove_thm (`lemma77`,
  "!p q r. p Or (q Or r) = (p Or q) Or r",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Or] THEN BETA_TAC
    THEN GEN_TAC THEN BOOL_CASES_TAC "(q:time->bool) n"
    THEN REWRITE_TAC[]);;

let lemma78 = prove_thm (`lemma78`,
  "!p q r. p And (q Or r) = (p And q) Or (p And r)",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[And;Or] THEN BETA_TAC
    THEN GEN_TAC THEN BOOL_CASES_TAC "(p:time->bool) n"
    THEN REWRITE_TAC[]);;

let lemma79 = prove_thm (`lemma79`,
  "!p q r. ((p Or q) And r) = ((p And r) Or (q And r))",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[And;Or] THEN BETA_TAC
    THEN GEN_TAC
    THEN BOOL_CASES_TAC "(r:time->bool) n"
    THEN REWRITE_TAC[]);;

let lemma80 = prove_thm (`lemma80`,
  "!p q r. (p Or (q And r)) = ((p Or q) And (p Or r))",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[And;Or] THEN BETA_TAC
    THEN GEN_TAC
    THEN BOOL_CASES_TAC "(p:time->bool) n"
    THEN REWRITE_TAC[]);;

let lemma81 = prove_thm (`lemma81`,
  "!p q r. ((p And q) Or r) = ((p Or r) And (q Or r))",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[And;Or] THEN BETA_TAC
    THEN GEN_TAC THEN BOOL_CASES_TAC "(r:time->bool) n"
    THEN REWRITE_TAC[]);;

let lemma82 = prove_thm (`lemma82`,
  "!p. p And Not p = False",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[And;Not;False]
    THEN BETA_TAC THEN GEN_TAC
    THEN BOOL_CASES_TAC "(p:time->bool) n"
    THEN REWRITE_TAC[]);;

let lemma83 = prove_thm (`lemma83`,
  "!p. False And p = False",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[And;False]
    THEN BETA_TAC);;
```

```
let lemma84 = prove_thm (`lemma84`,
  "!p. p And False = False",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[And;False] THEN BETA_TAC);;

let lemma85 = prove_thm (`lemma85`,
  "!p. True And p = p",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[And;True]
    THEN BETA_TAC THEN GEN_TAC THEN REFL_TAC);;

let lemma86 = prove_thm (`lemma86`,
  "!p. p And True = p",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[And;True]
    THEN BETA_TAC THEN GEN_TAC THEN REFL_TAC);;

let lemma87 = prove_thm (`lemma87`,
  "!p. p Or Not p = True",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Or;Not;True]
    THEN BETA_TAC THEN REWRITE_TAC[EXCLUDED_MIDDLE]);;

let lemma88 = prove_thm (`lemma88`,
  "!p. True Or p = True",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Or;True] THEN BETA_TAC);;

let lemma89 = prove_thm (`lemma89`,
  "!p. p Or True = True",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Or;True] THEN BETA_TAC);;
```

```
let lemma90 = prove_thm (`lemma90`,
  "!p. False Or p = p",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Or;False] THEN BETA_TAC
    THEN GEN_TAC THEN REFL_TAC);;

let lemma91 = prove_thm (`lemma91`,
  "!p. p Or False = p",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Or;False] THEN BETA_TAC
    THEN GEN_TAC THEN REFL_TAC);;

let lemma92 = prove_thm (`lemma92`,
  "!p q r w. (p And q) And (r And w) =
       (p And r) And (q And w)",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[And] THEN BETA_TAC
    THEN GEN_TAC
    THEN BOOL_CASES_TAC "(r:time->bool) n"
    THEN REWRITE_TAC[]
    THEN BOOL_CASES_TAC "(q:time->bool) n"
    THEN REWRITE_TAC[]);;

let lemma93 = prove_thm (`lemma93`,
  "!p. (p-->False) = Not p",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[And;Not;Imp;False] THEN BETA_TAC);;

let lemma94 = prove_thm (`lemma94`,
  "!p q. Valid(p-->q) /\ Valid p ==> Valid q",
    REPEAT GEN_TAC THEN REWRITE_TAC[Valid;Imp]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN ASSUM_LIST (\t. MATCH_MP_TAC (el 2 t))
    THEN ASM_REWRITE_TAC[]);;
```

```
let lemma95 = prove_thm (`lemma95`,
  "!p. Not(Sometime p) = Always(Not p)",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Sometime;Always;Not] THEN BETA_TAC
    THEN CONV_TAC (DEPTH_CONV NOT_EXISTS_CONV)
    THEN GEN_TAC THEN EQ_TAC THEN REPEAT STRIP_TAC
    THEN RES_TAC);;

let lemma96 = prove_thm (`lemma96`,
  "!p. Not(Always p) = Sometime(Not p)",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Sometime;Always;Not]
    THEN BETA_TAC THEN ONCE_REWRITE_TAC[NOT_EQ_REFL]
    THEN CONV_TAC (DEPTH_CONV NOT_EXISTS_CONV)
    THEN GEN_TAC THEN EQ_TAC
    THENL [REPEAT STRIP_TAC THEN RES_TAC THEN RES_TAC;
           REPEAT STRIP_TAC THEN RES_TAC
           THEN POP_ASSUM (ASSUME_TAC o(\th.
                  REWRITE_RULE[IMP_DISJ_THM] th))
           THEN ASM_REWRITE_TAC[]]);;

let lemma97 = prove_thm (`lemma97`,
  "!p q. Valid (Always(p-->q) --> (Always p --> Always q))",
    REPEAT GEN_TAC THEN REWRITE_TAC[Imp;Always;Valid]
    THEN BETA_TAC THEN REWRITE_TAC[IMP_AND]
    THEN REPEAT STRIP_TAC THEN RES_TAC);;

let lemma98 = prove_thm (`lemma98`,
  "!p. Valid (Always p --> p)",
    GEN_TAC THEN REWRITE_TAC[Imp;Always;Valid]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN POP_ASSUM MATCH_MP_TAC
    THEN REWRITE_TAC[LESS_EQ_REFL]);;
```

```
let lemma99 = prove_thm ('lemma99',
  "!p. Not(Next p) = Next(Not p)",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Not;Next]
    THEN BETA_TAC THEN GEN_TAC THEN REFL_TAC);;

let lemma100 = prove_thm ('lemma100',
  "!p q. Valid (Next(p-->q) --> (Next p --> Next q))",
    REPEAT GEN_TAC THEN REWRITE_TAC[Valid;Next;Imp]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[]);;

let lemma101 = prove_thm ('lemma101',
  "!p. Valid (Always p --> Next p)",
    GEN_TAC THEN REWRITE_TAC[Valid;Always;Next;Imp]
    THEN REPEAT STRIP_TAC THEN BETA_TAC
    THEN CONV_TAC LEFT_IMP_FORALL_CONV
    THEN EXISTS_TAC "SUC t"
    THEN REWRITE_TAC[LESS_EQ_SUC_REFL]);;

let lemma102 = prove_thm ('lemma102',
  "!p. Valid (Always p --> Next(Always p))",
    GEN_TAC THEN REWRITE_TAC[Valid;Next;Imp;Always]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN ASSUM_LIST (\t. MATCH_MP_TAC (el 2 t))
    THEN MATCH_MP_TAC (
         SPECL ["t:num";"SUC t";"t1:num"] LESS_EQ_TRANS)
    THEN ASM_REWRITE_TAC[LESS_EQ_SUC_REFL]);;

let lemma103 = prove_thm ('lemma103',
  "!p. Valid (Always(p-->Next p) --> (p-->Always p))",
    GEN_TAC THEN REWRITE_TAC[Valid;Always;Next;Imp]
    THEN BETA_TAC THEN INDUCT_TAC
    THENL [REWRITE_TAC[ZERO_LESS_EQ;IMP_AND]
           THEN ASSUME_TAC(
             ONCE_REWRITE_RULE[CONJ_SYM] INDUCTION)
           THEN ASM_REWRITE_TAC[];
           STRIP_TAC THEN STRIP_TAC THEN INDUCT_TAC
           THENL [REWRITE_TAC[LESS_OR_EQ;NOT_LESS_0;NOT_SUC];
           REWRITE_TAC[LESS_OR_EQ;LESS_MONO_EQ;INV_SUC_EQ]
                  THEN ASM_CASES_TAC "(SUC t)<=t1"
                  THEN RES_TAC THEN ASM_REWRITE_TAC[]
                  THEN ASSUME_TAC LESS_EQ THEN RES_TAC
                  THEN ASM_REWRITE_TAC[] THEN STRIP_TAC
                  THENL [ASSUM_LIST (\th. ASSUME_TAC
                   (REWRITE_RULE[(el 9 th)] (el 6 th)))
                    THEN RES_TAC;
                    POP_ASSUM MP_TAC
                    THEN CONV_TAC (ONCE_DEPTH_CONV SYM_CONV)
                      THEN STRIP_TAC THEN ASM_REWRITE_TAC[];
                       ASSUM_LIST (\th. ASSUME_TAC
                       (REWRITE_RULE[(el 7 th)] (el 4 th)))
                     THEN RES_TAC;
                     POP_ASSUM MP_TAC
                     THEN CONV_TAC (ONCE_DEPTH_CONV SYM_CONV)
                         THEN STRIP_TAC
                         THEN ASM_REWRITE_TAC[]]]]);;


let lemma104 = prove_thm (`lemma104`,
  "!p q. p Until q = q Or (p And Next(p Until q))",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Until;Or;And;Next]
    THEN BETA_TAC THEN GEN_TAC THEN EQ_TAC
    THEN STRIP_TAC
    THENL [ASM_CASES_TAC "n<t1"
           THENL [MATCH_MP_TAC OR_INTRO_THM2
             THEN ASSUM_LIST
(\th.ASM_REWRITE_TAC[REWRITE_RULE
                    [LESS_EQ_REFL;(el 1 th)]
                    (SPEC "n:num" (el 2 th))])
            THEN EXISTS_TAC "t1:num"
            THEN ASSUME_TAC LESS_EQ
            THEN RES_TAC THEN ASM_REWRITE_TAC[]
            THEN REPEAT STRIP_TAC
            THEN ASSUM_LIST (\th. ASSUME_TAC (REWRITE_RULE
               [LESS_OR_EQ;(el 10 th);(el 7 th)]
                (SPEC "t2:num" (el 8 th))))
            THEN RES_TAC THEN RES_TAC;
              ASSUM_LIST (\th. ASM_REWRITE_TAC[REWRITE_RULE
              [LESS_OR_EQ;(el 1 th)] (el 4 th)])];
           EXISTS_TAC "n:num"
           THEN ASM_REWRITE_TAC[LESS_EQ_REFL]
           THEN ONCE_REWRITE_TAC[CONJ_SYM]
           THEN REWRITE_TAC[LESS_EQ_ANTISYM];
           EXISTS_TAC "t1:num" THEN IMP_RES_TAC LESS_EQ
           THEN ASM_REWRITE_TAC[LESS_OR_EQ]
           THEN REPEAT STRIP_TAC
           THENL [ASSUME_TAC LESS_EQ THEN RES_TAC;
              POP_ASSUM (\t. ALL_TAC) THEN POP_ASSUM MP_TAC
              THEN CONV_TAC (ONCE_DEPTH_CONV SYM_CONV)
              THEN STRIP_TAC THEN ASM_REWRITE_TAC[]]]);;

let lemma105 = prove_thm (`lemma105`,
  "!p q. Valid ((p Until q) --> Sometime q)",
    REPEAT GEN_TAC THEN REWRITE_TAC[Valid;Until;Sometime;Imp]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "t1:num" THEN ASM_REWRITE_TAC[]);;

let lemma106 = prove_thm (`lemma106`,
  "!p q. p-->q = ((Not q) --> (Not p))",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Imp;Not] THEN BETA_TAC
    THEN REWRITE_TAC[IMP_DISJ_THM]
    THEN GEN_TAC THEN EQ_TAC THEN REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[]);;

let lemma107 = prove_thm ('lemma107',
  "!p. Not(Not p) = p",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Not] THEN BETA_TAC
    THEN REWRITE_TAC[]);;

let lemma108 = prove_thm ('lemma108',
  "!p. Valid p ==> Valid(Always p)",
    GEN_TAC THEN REWRITE_TAC[Valid;Always]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[]);;

let lemma109 = prove_thm ('lemma109',
  "!p. Valid p ==> Valid(Next p)",
    GEN_TAC THEN REWRITE_TAC[Valid;Next]
    THEN BETA_TAC THEN STRIP_TAC THEN INDUCT_TAC
    THEN ASM_REWRITE_TAC[]);;

let lemma110 = prove_thm ('lemma110',
  "!p. Valid (p-->Sometime p)",
    GEN_TAC THEN REWRITE_TAC[Valid;Sometime;Imp]
    THEN BETA_TAC THEN REPEAT STRIP_TAC THEN EXISTS_TAC "t"
    THEN ASM_REWRITE_TAC[LESS_EQ_REFL]);;

let lemma111 = prove_thm ('lemma111',
  "!p. Valid p ==> Valid(Sometime p)",
    GEN_TAC THEN REWRITE_TAC[Valid;Sometime]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "t"
    THEN ASM_REWRITE_TAC[LESS_EQ_REFL]);;

let lemma112 = prove_thm ('lemma112',
  "!p. Valid ((Next p) --> Sometime p)",
    GEN_TAC THEN REWRITE_TAC[Valid;Next;Sometime;Imp]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "SUC t"
    THEN ASM_REWRITE_TAC[LESS_EQ_SUC_REFL]);;

let lemma113 = prove_thm ('lemma113',
  "!p q. Valid (p-->q) ==> Valid(Always p --> Always q)",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Imp;Always]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN RES_TAC THEN RES_TAC);;

let lemma114 = prove_thm ('lemma114',
  "!p q. Valid (p-->q) ==> Valid(Sometime p --> Sometime q)",
    REPEAT GEN_TAC THEN REWRITE_TAC[Valid;Sometime;Imp]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN RES_TAC THEN EXISTS_TAC "t'"
    THEN ASM_REWRITE_TAC[]);;

let lemma115 = prove_thm ('lemma115',
  "!p q. Valid (p-->q) ==> Valid(Next p -->Next q)",
    REPEAT GEN_TAC THEN REWRITE_TAC[Valid;Imp;Next]
    THEN BETA_TAC THEN STRIP_TAC
    THEN ASM_REWRITE_TAC[]);;

let lemma116 = prove_thm ('lemma116',
  "!p. Valid (p-->Next p) ==> Valid(p-->Always p)",
    GEN_TAC THEN REWRITE_TAC[Valid;Always;Next;Imp]
    THEN BETA_TAC THEN STRIP_TAC THEN INDUCT_TAC
    THENL [REWRITE_TAC[ZERO_LESS_EQ]
```

```
          THEN POP_ASSUM MP_TAC
          THEN REWRITE_TAC[IMP_AND]
          THEN ASSUME_TAC(
             ONCE_REWRITE_RULE[CONJ_SYM] INDUCTION)
          THEN ASM_REWRITE_TAC[];
          STRIP_TAC THEN INDUCT_TAC
          THENL [REWRITE_TAC[LESS_OR_EQ;NOT_LESS_0;NOT_SUC];
            REWRITE_TAC[LESS_OR_EQ;
                 LESS_MONO_EQ;INV_SUC_EQ]
            THEN ASM_CASES_TAC "(SUC t)<=t1" THEN RES_TAC
            THEN ASSUME_TAC LESS_EQ THEN RES_TAC
            THEN ASM_REWRITE_TAC[] THEN STRIP_TAC
            THENL [ASSUME_TAC SUC_LESS_SUC
                  THEN RES_TAC THEN RES_TAC;
                  ASSUM_LIST (\th. ASM_REWRITE_TAC[
                    REWRITE_RULE [(el 1 th)] (el 8 th)]);
                  RES_TAC THEN RES_TAC;
                  ASSUM_LIST (\th. ASM_REWRITE_TAC[
                  REWRITE_RULE [(el 1 th)] (el 7 th)])]]]);;

let lemma117 = prove_thm (`lemma117`,
  "!p. Always p = Always (Always p)",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Always] THEN BETA_TAC
    THEN GEN_TAC THEN EQ_TAC
    THENL [REPEAT STRIP_TAC THEN ASSUME_TAC LESS_EQ_TRANS
          THEN RES_TAC THEN RES_TAC;
          REPEAT STRIP_TAC
          THEN ASSUME_TAC (SPEC "n:num" LESS_EQ_REFL)
          THEN RES_TAC]);;

let lemma118 = prove_thm (`lemma118`,
  "!p. Sometime p = Sometime (Sometime p)",
    GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Sometime] THEN BETA_TAC
    THEN GEN_TAC THEN EQ_TAC
    THENL [REPEAT STRIP_TAC THEN EXISTS_TAC "t'"
           THEN ASM_REWRITE_TAC[] THEN EXISTS_TAC "t'"
           THEN ASM_REWRITE_TAC[LESS_EQ_REFL];
           STRIP_TAC THEN EXISTS_TAC "t'':num"
           THEN ASSUME_TAC LESS_EQ_TRANS
           THEN RES_TAC THEN ASM_REWRITE_TAC[]]);;

let lemma119 = prove_thm (`lemma119`,
  "!p q. Valid(Always(p-->q))
            ==> Valid(Sometime p --> Sometime q)",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Always;Imp;Sometime]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN RES_TAC THEN EXISTS_TAC "t'"
    THEN ASM_REWRITE_TAC[]);;

let lemma120 = prove_thm (`lemma120`,
  "!p q. Always(p And q) = (Always p) And (Always q)",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Always;And] THEN BETA_TAC
    THEN GEN_TAC THEN EQ_TAC THEN REPEAT STRIP_TAC
    THEN RES_TAC);;

let lemma121 = prove_thm (`lemma121`,
  "!p q. Sometime(p Or q) = (Sometime p) Or (Sometime q)",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Sometime;Or] THEN BETA_TAC
    THEN GEN_TAC THEN EQ_TAC
    THENL [REPEAT STRIP_TAC THEN CONV_TAC OR_EXISTS_CONV
           THEN EXISTS_TAC "t'" THEN ASM_REWRITE_TAC[];
           STRIP_TAC THEN EXISTS_TAC "t'"
           THEN ASM_REWRITE_TAC[]]);;
```

```
let lemma122 = prove_thm ('lemma122',
  "!p q. Valid ((Always p Or Always q) --> Always(p Or q))",
    REPEAT GEN_TAC THEN REWRITE_TAC[Valid;Always;Or;Imp]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN RES_TAC THEN ASM_REWRITE_TAC[]);;

let lemma123 = prove_thm ('lemma123',
  "!p q. Valid (Sometime(p And q)
             --> (Sometime p And Sometime q))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Sometime;Imp;And]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC"t'" THEN ASM_REWRITE_TAC[]);;

let lemma124 = prove_thm ('lemma124',
  "!p q. Valid (((Always p) And (Sometime q))
        --> Sometime(p And q))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Always;And;Imp;Sometime]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN RES_TAC THEN EXISTS_TAC "t'"
    THEN ASM_REWRITE_TAC[]);;

let lemma125 = prove_thm ('lemma125',
  "!p q. Next(p And q) = ((Next p) And (Next q))",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Next;And] THEN BETA_TAC
    THEN GEN_TAC THEN REFL_TAC);;

let lemma126 = prove_thm ('lemma126',
  "!p q. Next(p Or q) = ((Next p) Or (Next q))",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Next;Or] THEN BETA_TAC
    THEN GEN_TAC THEN REFL_TAC);;


let lemma127 = prove_thm (`lemma127`,
  "!p q. Next(p-->q) = ((Next p)-->(Next q))",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Imp;Next] THEN BETA_TAC
    THEN GEN_TAC THEN REFL_TAC);;


let lemma128 = prove_thm (`lemma128`,
  "!p q. Next(p Equiv q) = ((Next p) Equiv (Next q))",
    REPEAT GEN_TAC THEN CONV_TAC FUN_EQ_CONV
    THEN REWRITE_TAC[Next;Equiv] THEN BETA_TAC
    THEN GEN_TAC THEN REFL_TAC);;


let lemma129 = prove_thm (`lemma129`,
  "!p. Valid(p And (Sometime(Not p)))
              ==> Valid(Sometime(p And (Next p)))",
    GEN_TAC
    THEN REWRITE_TAC[Valid;And;Sometime;Not;Imp;Next]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "t" THEN ASM_REWRITE_TAC[LESS_EQ_REFL]);;


let lemma130 = prove_thm (`lemma130`,
  "!p1 p2 q1 q2.
     (Valid (p1-->p2)) /\ (Valid(p2-->(Always q1)))
        /\ (Valid(q1-->q2)) ==> (Valid(p1-->(Always q2)))",
    REPEAT GEN_TAC THEN REWRITE_TAC[Valid;Imp;Always]
    THEN BETA_TAC THEN REPEAT STRIP_TAC THEN RES_TAC
    THEN RES_TAC THEN RES_TAC);;


let lemma131 = prove_thm (`lemma131`,
  "!p1 p2 q1 q2.
     (Valid (p1-->p2)) /\ (Valid(p2-->(Sometime q1)))
      /\ (Valid(q1-->q2)) ==> (Valid(p1-->(Sometime q2)))",
    REPEAT GEN_TAC THEN REWRITE_TAC[Valid;Imp;Sometime]
    THEN BETA_TAC THEN REPEAT STRIP_TAC THEN RES_TAC
    THEN RES_TAC THEN EXISTS_TAC "t'"
    THEN ASM_REWRITE_TAC[] THEN RES_TAC);;

let lemma132 = prove_thm (`lemma132`,
  "!p1 p2 q1 q2.
      (Valid (p1-->p2)) /\ (Valid(p2-->(Next q1)))
         /\ (Valid(q1-->q2)) ==> (Valid(p1-->(Next q2)))",
    REPEAT GEN_TAC THEN REWRITE_TAC[Valid;Imp;Next]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN RES_TAC THEN RES_TAC THEN RES_TAC);;

let lemma133 = prove_thm (`lemma133`,
  "!p q r. Valid((p-->Always q) And (q-->Always r))
                  ==> Valid(p-->(Always r))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Always;Imp;And]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN RES_TAC THEN RES_TAC THEN POP_ASSUM MP_TAC
    THEN CONV_TAC(DEPTH_CONV LEFT_IMP_FORALL_CONV)
    THEN EXISTS_TAC "t'"
    THEN REWRITE_TAC[LESS_EQ_REFL]);;

let lemma134 = prove_thm (`lemma134`,
  "!p q r. Valid ((p-->Sometime q) And (q-->Sometime r))
                 ==> Valid(p-->Sometime r)",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Imp;Sometime;And]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN RES_TAC THEN RES_TAC
    THEN ASSUME_TAC(SPECL["t:num";"t':num";"t''':num"]
         LESS_EQ_TRANS)
    THEN RES_TAC THEN EXISTS_TAC "t''':num"
    THEN ASM_REWRITE_TAC[]);;

let lemma135 = prove_thm ('lemma135',
  "!p q r. Valid(
       (p-->Sometime q) And (r-->(q And (p And (Next r)))))
              ==> Valid(r-->(p Until q))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Imp;Sometime;Or;And;Next;Until]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "t" THEN REWRITE_TAC[LESS_EQ_REFL]
    THEN RES_TAC THEN ASM_REWRITE_TAC[]
    THEN REPEAT STRIP_TAC
    THEN ASSUME_TAC(
         SPECL["t2:num";"t:num"]LESS_IMP_LESS_OR_EQ)
    THEN ASSUM_LIST(\th. ASSUME_TAC(
         REWRITE_RULE [(el 2 th)](el 1 th)))
    THEN ASSUM_LIST (\th. ASSUME_TAC (REWRITE_RULE
         [(el 4 th);(el 1 th)] (SPECL["t:num";"t2:num"]
         LESS_EQUAL_ANTISYM)))
    THEN ASSUM_LIST (\th. ASM_REWRITE_TAC [
         REWRITE_RULE [(el 1 th)] (el 7 th)]));;

let lemma136 = prove_thm ('lemma136',
  "!p1 p2 q1 q2. (Valid(p1-->p2)) /\ (Valid(q1-->q2)) ==>
       (Valid ((p1 Until q1) --> (p2 Until q2)))",
    REPEAT GEN_TAC THEN REWRITE_TAC[Valid;Imp;Until]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "t1" THEN RES_TAC
    THEN ASM_REWRITE_TAC[] THEN REPEAT STRIP_TAC
    THEN RES_TAC THEN RES_TAC);;

let lemma137 = prove_thm ('lemma137',
  "!p q.
     Valid (((Always p) And (Sometime q)) --> (p Until q))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Always;And;Sometime;Until;Imp]
    THEN BETA_TAC THEN REPEAT STRIP_TAC THEN EXISTS_TAC "t'"
    THEN ASM_REWRITE_TAC[] THEN REPEAT STRIP_TAC
    THEN RES_TAC);;

let lemma138 = prove_thm (`lemma138`,
  "!p q r. Valid (((Always p) And (q Until r)) -->
              ((p And q) Until (p And r)))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Until;Valid;Imp;Always;And]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "t1" THEN RES_TAC
    THEN ASM_REWRITE_TAC[] THEN REPEAT STRIP_TAC
    THEN RES_TAC);;

let lemma139 = prove_thm (`lemma139`,
  "!p q r. Valid ((p Until (q And r))
          --> ((p Until q) And (p Until r)))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[Valid;Until;And;Imp]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "t1:num" THEN ASM_REWRITE_TAC[]);;

let lemma140 = prove_thm (`lemma140`,
  "!p q r.
      Valid ((p Until (q And r))-->((p Until q) Until r))",
    REPEAT GEN_TAC THEN REWRITE_TAC[Valid;Until;And;Imp]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC"t1" THEN REPEAT STRIP_TAC
    THENL [ASM_REWRITE_TAC[];
           ASM_REWRITE_TAC[];
           EXISTS_TAC "t1"
           THEN ASSUM_LIST(\th.
          ASSUME_TAC(REWRITE_RULE[(el 1 th)]
          (SPECL ["t2:num";"t1:num"]LESS_IMP_LESS_OR_EQ)))
        THEN ASM_REWRITE_TAC[] THEN REPEAT STRIP_TAC
        THEN ASSUM_LIST(\th.ASSUME_TAC(
          REWRITE_RULE[(el 5 th);(el 2 th)]
    (SPECL["t:num";"t2:num";"t2':num"]LESS_EQ_TRANS)))
   THEN RES_TAC]);;
%<------------------------------------------------------------>
```

Annex C: HOL Specification of DBW Controller

C.1 Definitions

Mode_TY_DEF

$\vdash \exists rep.\ \mathsf{TYPE\_DEFINITION}\ (\mathsf{TRP}\ (\lambda v\ tl.\ (v = \mathsf{INL\ one}) \wedge (\mathsf{LENGTH}\ tl = 0) \vee$
$(v = \mathsf{INR\ (INL\ one)}) \wedge (\mathsf{LENGTH}\ tl = 0) \vee (v = \mathsf{INR\ (INR\ (INL\ one))}) \wedge$
$(\mathsf{LENGTH}\ tl = 0) \vee (v = \mathsf{INR\ (INR\ (INR\ (INL\ one)))}) \wedge (\mathsf{LENGTH}\ tl = 0) \vee$
$(v = \mathsf{INR\ (INR\ (INR\ (INR\ (INL\ one))))}) \wedge (\mathsf{LENGTH}\ tl = 0) \vee$
$(v = \mathsf{INR\ (INR\ (INR\ (INR\ (INR\ one))))}) \wedge (\mathsf{LENGTH}\ tl = 0)))\ rep$

Mode_ISO_DEF $\vdash (\forall a.\ \mathsf{ABS\_Mode}\ (\mathsf{REP\_Mode}\ a) = a) \wedge (\forall r.\ \mathsf{TRP}\ (\lambda v\ tl.\ (v = \mathsf{INL\ one}) \wedge$
$(\mathsf{LENGTH}\ tl = 0) \vee (v = \mathsf{INR\ (INL\ one)}) \wedge (\mathsf{LENGTH}\ tl = 0) \vee$
$(v = \mathsf{INR\ (INR\ (INL\ one))}) \wedge (\mathsf{LENGTH}\ tl = 0) \vee$
$(v = \mathsf{INR\ (INR\ (INR\ (INL\ one)))}) \wedge (\mathsf{LENGTH}\ tl = 0) \vee$
$(v = \mathsf{INR\ (INR\ (INR\ (INR\ (INL\ one))))}) \wedge (\mathsf{LENGTH}\ tl = 0) \vee$
$(v = \mathsf{INR\ (INR\ (INR\ (INR\ (INR\ one))))}) \wedge$
$(\mathsf{LENGTH}\ tl = 0))\ r = (\mathsf{REP\_Mode}\ (\mathsf{ABS\_Mode}\ r) = r))$

Reset_DEF $\vdash \mathsf{Reset} = \mathsf{ABS\_Mode}\ (\mathsf{Node}\ (\mathsf{INL\ one})\ [\,])$

Idle_DEF $\vdash \mathsf{Idle} = \mathsf{ABS\_Mode}\ (\mathsf{Node}\ (\mathsf{INR\ (INL\ one)})\ [\,])$

Traction_DEF $\vdash \mathsf{Traction} = \mathsf{ABS\_Mode}\ (\mathsf{Node}\ (\mathsf{INR\ (INR\ (INL\ one))})\ [\,])$

Shutdown_DEF $\vdash \mathsf{Shutdown} = \mathsf{ABS\_Mode}\ (\mathsf{Node}\ (\mathsf{INR\ (INR\ (INR\ (INL\ one)))})\ [\,])$

Manual_DEF $\vdash \mathsf{Manual} = \mathsf{ABS\_Mode}\ (\mathsf{Node}\ (\mathsf{INR\ (INR\ (INR\ (INR\ (INL\ one))))})\ [\,])$

Cruise_DEF $\vdash \mathsf{Cruise} = \mathsf{ABS\_Mode}\ (\mathsf{Node}\ (\mathsf{INR\ (INR\ (INR\ (INR\ (INR\ one))))})\ [\,])$

RUNNING_DEF $\vdash \forall s.\ \mathsf{RUNNING}\ s = \mathsf{FST}\ s$

GEAR_DEF $\vdash \forall s.$ GEAR s = FST (SND s)

BRAKE_DEF $\vdash \forall s.$ BRAKE s = FST (SND (SND s))

WATCHOUT_DEF $\vdash \forall s.$ WATCHOUT s = FST (SND (SND (SND s)))

WFAILED_DEF $\vdash \forall s.$ WFAILED s = FST (SND (SND (SND (SND s))))

OVERTEMP_DEF $\vdash \forall s.$ OVERTEMP s = FST (SND (SND (SND (SND (SND s)))))

IDLE_DEF $\vdash \forall s.$ IDLE s = FST (SND (SND (SND (SND (SND (SND s))))))

TRACTION_DEF $\vdash \forall s.$ TRACTION s =
FST (SND (SND (SND (SND (SND (SND (SND s)))))))

CRUISE_DEF $\vdash \forall s.$ CRUISE s =
FST (SND (SND (SND (SND (SND (SND (SND (SND s))))))))

RESUME_DEF $\vdash \forall s.$ RESUME s = FST (SND (SND (SND (SND
(SND (SND (SND (SND (SND s)))))))))

DEMAND_DEF $\vdash \forall s.$ DEMAND s = FST (SND (SND (SND (SND (SND
(SND (SND (SND (SND (SND s))))))))))

SPEED_DEF $\vdash \forall s.$ SPEED s = FST (SND (SND (SND (SND (SND
(SND (SND (SND (SND (SND (SND s)))))))))))

FBANG_DEF $\vdash \forall s.$ FBANG s = SND (SND (SND (SND (SND (SND
(SND (SND (SND (SND (SND (SND s)))))))))))

R_COND_DEF $\vdash \forall running.$ R_COND $running$ = $(\lambda t.$ Not $running\ t)$

M_COND_DEF $\vdash \forall mode\ idle\ brake\ gear.$
I_COND $(mode, idle, brake, gear)$ =
$(\lambda t. ((mode =$ Manual$) \vee (mode =$ Cruise$) \vee (mode =$ Idle$)) \wedge idle\ t\ \wedge$
$(brake\ t \vee \neg gear\ t))$

S_COND_DEF $\vdash \forall watchout\ overtemp\ stop.$
S_COND $(watchout, overtemp, stop)$ =
$(\lambda t. watchout\ t \vee overtemp\ t \vee stop)$

C_COND_DEF $\vdash \forall cruise\ speed\ resume\ valid\_cruise\ gear\ mode\ demand\ brake.$
C_COND $(cruise, speed, resume, valid\_cruise, gear, mode, demand, brake)$ =
$(\lambda t. (cruise\ t \wedge \neg speed$ Eq 0 $t \wedge (mode =$ Manual$) \vee$

$\neg(speed\ t < demand\ t)\ \wedge\ \neg brake\ t\ \wedge\ (mode = \mathsf{Cruise})\ \vee\ resume\ t\ \wedge$
$valid\_cruise\ \wedge\ (mode = \mathsf{Idle}))\ \wedge\ gear\ t)$

T_COND_DEF $\vdash \forall mode\ traction\ speed\ gear\ idle.$
$\mathsf{T\_COND}\ (mode, traction, speed, gear, idle) =$
$(\lambda t.((mode = \mathsf{Manual})\ \vee\ (mode = \mathsf{Traction}))\ \wedge\ traction\ t\ \wedge$
$speed\ \mathsf{Eq}\ 0\ t\ \wedge\ \neg gear\ t\ \wedge\ \neg idle\ t)$

Dir_TY_DEF $\vdash \exists rep.\ \mathsf{TYPE\_DEFINITION}\ (\mathsf{TRP}\ (\lambda v\ tl.(v = \mathsf{INL\ one}) \wedge (\mathsf{LENGTH}\ tl = 0) \vee$
$(v = \mathsf{INR}\ (\mathsf{INL\ one}))\ \wedge\ (\mathsf{LENGTH}\ tl = 0)\ \vee\ (v = \mathsf{INR}\ (\mathsf{INR}\ (\mathsf{INL\ one})))\ \wedge$
$(\mathsf{LENGTH}\ tl = 0)\ \vee\ (v = \mathsf{INR}\ (\mathsf{INR}\ (\mathsf{INR\ one})))\ \wedge\ (\mathsf{LENGTH}\ tl = 0)))\ rep$

Dir_ISO_DEF $\vdash (\forall a.\ \mathsf{ABS\_Dir}\ (\mathsf{REP\_Dir}\ a) = a)\ \wedge\ (\forall r.\ \mathsf{TRP}\ (\lambda v\ tl.(v = \mathsf{INL\ one})\ \wedge$
$(\mathsf{LENGTH}\ tl = 0)\ \vee\ (v = \mathsf{INR}\ (\mathsf{INL\ one}))\ \wedge\ (\mathsf{LENGTH}\ tl = 0)\ \vee$
$(v = \mathsf{INR}\ (\mathsf{INR}\ (\mathsf{INL\ one})))\ \wedge\ (\mathsf{LENGTH}\ tl = 0)\ \vee$
$(v = \mathsf{INR}\ (\mathsf{INR}\ (\mathsf{INR\ one})))\ \wedge$
$(\mathsf{LENGTH}\ tl = 0))\ r = (\mathsf{REP\_Dir}\ (\mathsf{ABS\_Dir}\ r) = r))$

Close_DEF $\vdash \mathsf{Close} = \mathsf{ABS\_Dir}\ (\mathsf{Node}\ (\mathsf{INL\ one})\ [\,])$

Forward_DEF $\vdash \mathsf{Forward} = \mathsf{ABS\_Dir}\ (\mathsf{Node}\ (\mathsf{INR}\ (\mathsf{INL\ one}))\ [\,])$

Backward_DEF $\vdash \mathsf{Backward} = \mathsf{ABS\_Dir}\ (\mathsf{Node}\ (\mathsf{INR}\ (\mathsf{INR}\ (\mathsf{INL\ one})))\ [\,])$

Neutral_DEF $\vdash \mathsf{Neutral} = \mathsf{ABS\_Dir}\ (\mathsf{Node}\ (\mathsf{INR}\ (\mathsf{INR}\ (\mathsf{INR\ one})))\ [\,])$

PWM_DEF $\vdash \forall pwm\_act\ ang\ dem.$
$\mathsf{PWM}\ (pwm\_act, ang, dem) =$
$((dem = 0) \Rightarrow \mathsf{Close}\ |$
$((dem > ang) \Rightarrow ((pwm\_act = \mathsf{Forward}) \Rightarrow \mathsf{Neutral}\ |\ \mathsf{Forward})\ |$
$((ang > dem) \Rightarrow ((pwm\_act = \mathsf{Backward}) \Rightarrow \mathsf{Neutral}\ |\ \mathsf{Backward})\ |$
$\mathsf{Neutral})))$

ILLEGAL $\vdash \forall b\ p.\ \mathsf{ILLEGAL}\ b\ p = (\lambda t.\ \neg\mathsf{Xor2}\ (p, b)\ t)$

DecodeSig_DEF $\vdash \forall sig\ runningt\ geart\ braket\ watchoutt\ wfailedt\ overtempt$
$idlet\ tractiont\ cruiset\ resumet\ demandt\ speedt\ fbangt.$
$\mathsf{DecodeSig}\ (sig, runningt, geart, braket, watchoutt, wfailedt,$
$overtempt, idlet, tractiont, cruiset, resumet, demandt, speedt, fbangt) =$
$(\forall t.\ runningt\ t, geart\ t, braket\ t, watchoutt\ t, wfailedt\ t, overtempt\ t,$
$idlet\ t, tractiont\ t, cruiset\ t, resumet\ t, demandt\ t, speedt\ t, fbangt\ t = sig\ t)$

```
NextState_DEF ⊢ ∀s mode valid_cruise pwm_act.
        NextState (s, mode, valid_cruise, pwm_act) =
        (λt. let running = RUNNING s in
        (let gear = GEAR s in
        (let brake = BRAKE s in
        (let watchout = WATCHOUT s in
        (let wfailed = WFAILED s in
        (let overtemp = OVERTEMP s in
        (let idle = IDLE s in
        (let traction = TRACTION s in
        (let cruise = CRUISE s in
        (let resume = RESUME s in
        (let demand = DEMAND s in
        (let speed = SPEED s in
        (let fbang = FBANG s in
        (let stop_ok = ILLEGAL idle (Not (demand Eq 0)) t ∨ wfailed t ∨
        (mode = Shutdown)) in
        (let reset_ok = R_COND running t in
        (let cruise_ok = (¬reset_ok ∧
        C_COND (cruise, speed, resume, valid_cruise, gear, mode, demand, brake) t) in
        (let traction_ok = (¬reset_ok ∧
        T_COND (mode, traction, speed, gear, idle) t) in
        (let shutdown_ok = S_COND (watchout, overtemp, stop_ok) t in
        (let idle_ok = (¬reset_ok ∧
        I_COND (mode, idle, brake, gear) t) in
        (let valid_c = ((mode = Cruise) ⇒ ¬reset_ok | valid_cruise) in
        (reset_ok ⇒ (Reset, valid_c, PWM (pwm_act, fbang t, 0) |
        (shutdown_ok ⇒ (Shutdown, valid_c, PWM (pwm_act, fbang t, 0) |
        (idle_ok ⇒ (Idle, valid_c, PWM (pwm_act, fbang t, 1) |
        (cruise_ok ⇒ (Cruise, valid_c, PWM (pwm_act, fbang t, speed t) |
        (traction_ok ⇒ (Traction, valid_c, PWM (pwm_act, fbang t, demand t) |
        (Manual, F, PWM (pwm_act, fbang t, demand t))))))))))))))))))))))))))

BACK_BEHAV_DEF ⊢ ∀sigt modet valid_ct pwm_actt.
        BACK_BEHAV (sigt, modet, valid_ct, pwm_actt) =
```

$(\lambda t.\ \mathsf{NextState}\ (sigt\ t, modet\ t, valid\_ct\ t, pwm\_actt\ t)\ t =$

$modet\ (t+1), valid\_ct\ (t+1), pwm\_actt\ (t+1))$

`DecodeCrtl` $\vdash \forall pwm\_act\ forward\ backward\ neutral\ close.$

$\mathsf{DecodeCtrl}\ (pwm\_act, forward, backward, neutral, close) =$

$(\lambda t.\ (forward\ t = (pwm\_act = \mathsf{Forward}))\ \wedge$

$(backward\ t = (pwm\_act = \mathsf{Backward}))\ \wedge$

$(neutral\ t = (pwm\_act = \mathsf{Neutral}))\ \wedge$

$(close\ t = (pwm\_act = \mathsf{Close})))$

`Accelerate_DEF` $\vdash \forall act\_start\ forward.$

$\mathsf{Accelerate}\ (act\_start, forward) = act\_start\ \mathsf{And}\ forward$

`Decelerate_DEF` $\vdash \forall act\_start\ backward.$

$\mathsf{Decelerate}\ (act\_start, backward) = act\_start\ \mathsf{And}\ backward$

`RESPONSE_DEF` $\vdash \forall forward\ backward\ neutral\ output\ act\_start.$

$\mathsf{RESPONSE}\ (forward, backward, neutral, output, act\_start) =$

$(\lambda t.\ (forward\ t \Rightarrow (output\ \mathsf{Within\ Len}\ act\_start\ 10)\ \mathsf{When}\ act\_start\ |$

$(backward\ t \Rightarrow (output\ \mathsf{Within\ Len}\ act\_start\ 10)\ \mathsf{When}\ act\_start\ |$

$neutral\ t)))$

`REGISTER_DEF` $\vdash \forall act\_stop\ act\_start\ neutral.$

$\mathsf{REGISTER}\ (act\_stop, act\_start, neutral) =$

$(\lambda t.\ \mathsf{Always}\ ((act\_stop\ \mathsf{And\ Not}\ neutral)$

$\longrightarrow (act\_start\ \mathsf{Within\ Len}\ (\mathsf{Not}\ neutral)\ 5))\ t)$

`Below_DEF` $\vdash \forall pos\ ang.\ pos\ \mathsf{Below}\ ang = (\lambda t.\ pos\ \mathsf{Eq}\ (\mathsf{PRE}\ ang)\ t)$

`Above_DEF` $\vdash \forall pos\ ang.\ pos\ \mathsf{Above}\ ang = (\lambda t.\ pos\ \mathsf{Eq}\ (\mathsf{SUC}\ ang)\ t)$

`TRANSIT_DEF` $\vdash \forall accelerating\ decelerating.$

$\mathsf{TRANSIT}\ (accelerating, decelerating) =$

$(\lambda t.\ (\forall pos\ ang.\ \mathsf{Always}\ (((accelerating\ \mathsf{And\ Not}\ (pos\ \mathsf{Below}\ ang))$

$\longrightarrow (\mathsf{Not}\ (pos\ \mathsf{Below}\ ang)\ \mathsf{Within\ Len}\ accelerating\ 2))$

$\mathsf{And}\ ((decelerating\ \mathsf{And\ Not}\ (pos\ \mathsf{Above}\ ang))$

$\longrightarrow (\mathsf{Not}\ (pos\ \mathsf{Above}\ ang)\ \mathsf{Within\ Len}\ decelerating\ 2)))\ t))$

`POS_DIR_DEF` $\vdash \forall act\_stop\ accelerating\ decelerating\ close\ forward\ backward$
$neutral.\ \mathsf{POS\_DIR}\ (act\_stop, accelerating, decelerating,$

close, *forward*, *backward*, *neutral*) =

(λt. (*act_stop* $t \Rightarrow$ Next *close* t |

(*accelerating* $t \Rightarrow$ (*forward* $t \Rightarrow$ Next *forward* t |

(*backward* $t \Rightarrow$ Next (Not *forward*) t | Next *neutral* t)) |

(*decelerating* $t \Rightarrow$ (*forward* $t \Rightarrow$ Next (Not *forward*) t |

(*backward* $t \Rightarrow$ Next *backward* t | Next *neutral* t)) | Next *neutral* t))))

PERFORM1_DEF $\vdash \forall$*act_stop act_start accelerating decelerating forward backward neutral close*.

PERFORM1 (*act_stop*, *act_start*, *accelerating*, *decelerating*,

forward, *backward*, *neutral*, *close*) =

(λt. Always (*act_stop* —— Next (*neutral* $t \Rightarrow$ *act_stop* |

(*act_start* And (Not *decelerating*) $t \Rightarrow$ *forward* |

(*act_start* And (Not *accelerating*) $t \Rightarrow$ *backward* | *close*)))) t)

PERFORM2_DEF $\vdash \forall$*act_start neutral*. PERFORM2 (*act_start*, *neutral*) =

(λt. Always (*act_start* ⟶ Not *neutral*) t)

PERFORM3_DEF $\vdash \forall$*forward backward neutral act_stop act_start*.

PERFORM3 (*forward*, *backward*, *neutral*, *act_stop*, *act_start*) =

(λt. Always (*forward* $t \Rightarrow$ Next (Not *forward* $t \Rightarrow$ *act_stop* | *forward*) |

(*backward* $t \Rightarrow$ Next (Not *backward* $t \Rightarrow$ *act_stop* | *backward*) |

(*neutral* $t \Rightarrow$ Next (Not *neutral* $t \Rightarrow$ *act_start* | *act_stop*) | *act_stop*))) t)

PERFORM4_DEF $\vdash \forall$*forward backward neutral accelerating decelerating*.

PERFORM4 (*forward*, *backward*, *neutral*, *accelerating*, *decelerating*) =

(λt. Always (*forward* $t \Rightarrow$ Next (*accelerating* $t \Rightarrow$ *forward* | *neutral*) |

(*backward* $t \Rightarrow$ Next (*decelerating* $t \Rightarrow$ *backward* | *neutral*) | Next *neutral*)) t)

FORE_BEHAV_DEF $\vdash \forall$*pwm_act output*. FORE_BEHAV (*pwm_act*, *output*) =

(λt. ($\forall$*act_start act_stop close forward backward neutral*.

let *accelerating* = Accelerate (*act_start*, *forward*)**in**(**let***decelerating* =

Decelerate (*act_start*, *backward*) **in**

(DecodeCtrl (*pwm_act*, *forward*, *backward*, *neutral*, *close*)) And

(RESPONSE (*forward*, *backward*, *neutral*, *output*, *act_start*) And

(REGISTER (*act_stop*, *act_start*, *neutral*) And

(TRANSIT (*accelerating*, *decelerating*) And

(POS_DIR (*act_stop*, *accelerating*, *decelerating*, *close*,

$\mathit{forward}, \mathit{backward}, \mathit{neutral}$) And

(PERFORM1 ($\mathit{act\_stop}, \mathit{act\_start}, \mathit{accelerating}, \mathit{decelerating},$

$\mathit{forward}, \mathit{backward}, \mathit{neutral}, \mathit{close}$) And

(PERFORM2 ($\mathit{act\_start}, \mathit{neutral}$) And

(PERFORM3 ($\mathit{forward}, \mathit{backward}, \mathit{neutral}, \mathit{act\_stop}, \mathit{act\_start}$) And

(PERFORM4 ($\mathit{forward}, \mathit{backward}, \mathit{neutral}, \mathit{accelerating}, \mathit{decelerating}$))))))) t)))

DBW_BEHAV_DEF $\vdash$ DBW_BEHAV = ($\forall t\ \mathit{sigt}\ \mathit{modet}\ \mathit{valid_ct}\ \mathit{pwm_actt}\ \mathit{output}\ \mathit{timer}.$

Always ((BACK_BEHAV ($\mathit{sigt}, \mathit{modet}, \mathit{valid_ct}, \mathit{pwm_actt}$) Upon timer)

(FORE_BEHAV ($\mathit{pwm_actt}\ t, \mathit{output}$))) t $\wedge$ Valid timer)

State_Safe_DEF $\vdash \forall s\ m\ v\ p.$ State_Safe $(s, m, v, p) =$

$(\lambda t. \neg$WFAILED $s\ t\ \vee\ (p =$ Close))

SAFE_DEF $\vdash \forall s\ m\ v\ p.$ SAFE $(s, m, v, p) =$

$(\lambda t.$ State_Safe (s, NextState $(s, m, v, p)\ t)\ t)$

INIT_DEF $\vdash \forall s\ m\ v\ p.$ INIT $(s, m, v, p) =$

$(\lambda t. \neg$RUNNING $s\ t\ \wedge$ (DEMAND s) Eq 0 t)

ArbNot0_DEF $\vdash$ ArbNot0 = $(\lambda t. (\varepsilon n. \neg(n = 0)))$

Arb_DEF $\vdash$ Arb = $(\lambda t.$ ARB)

Zero_DEF $\vdash$ Zero = $(\lambda t. 0)$

Init_DEF $\vdash \forall ip\ op.$ Init $(ip, op) = (\forall t.$ INIT $(ip, op)\ t)$

Safe_DEF $\vdash \forall ip\ op.$ Safe $(ip, op) = (\forall t.$ SAFE $(ip, op)\ t)$

Nextstate_DEF $\vdash \forall s\ m\ v\ p\ s'\ m'\ v'\ p'.$

Nextstate $(s, m, v, p)\ (s', m', v', p') = (\forall t.$ NextState $(s, m, v, p)\ t = m', v', p')$

C.2 Theorems

Mode_Axiom $\vdash \forall e_0\ e_1\ e_2\ e_3\ e_4\ e_5. (\exists\forall fn. (fn$ Reset $= e_0) \wedge (fn$ Idle $= e_1) \wedge$

$(fn$ Traction $= e_2) \wedge (fn$ Shutdown $= e_3) \wedge (fn$ Manual $= e_4) \wedge$

$(fn$ Cruise $= e_5))$

Mode_Distinct $\vdash \neg$(Reset = Idle) $\wedge$ $\neg$(Reset = Traction) $\wedge$ $\neg$(Reset = Shutdown) $\wedge$

$\neg$(Reset = Manual) $\wedge$ $\neg$(Reset = Cruise) $\wedge$ $\neg$(Idle = Traction) $\wedge$

$\neg$(Idle = Shutdown) $\wedge$ $\neg$(Idle = Manual) $\wedge$ $\neg$(Idle = Cruise) $\wedge$

$\neg(\mathsf{Traction} = \mathsf{Shutdown}) \wedge \neg(\mathsf{Traction} = \mathsf{Manual}) \wedge \neg(\mathsf{Traction} = \mathsf{Cruise}) \wedge$
$\neg(\mathsf{Shutdown} = \mathsf{Manual}) \wedge \neg(\mathsf{Shutdown} = \mathsf{Cruise}) \wedge \neg(\mathsf{Manual} = \mathsf{Cruise})$

Mode_Induct $\vdash \forall P.\, P\ \mathsf{Reset} \wedge P\ \mathsf{Idle} \wedge P\ \mathsf{Traction} \wedge P\ \mathsf{Shutdown} \wedge P\ \mathsf{Manual} \wedge$
$P\ \mathsf{Cruise} \supset (\forall M.\, P\ M)$

Mode_Cases $\vdash \forall M.\,(M = \mathsf{Reset}) \vee (M = \mathsf{Idle}) \vee (M = \mathsf{Traction}) \vee (M = \mathsf{Shutdown}) \vee$
$(M = \mathsf{Manual}) \vee (M = \mathsf{Cruise})$

Dir_Axiom $\vdash \forall e_0\, e_1\, e_2\, e_3.\,(\exists! fn.\,(fn\ \mathsf{Close} = e_0) \wedge (fn\ \mathsf{Forward} = e_1) \wedge$
$(fn\ \mathsf{Backward} = e_2) \wedge (fn\ \mathsf{Neutral} = e_3))$

Dir_Distinct $\vdash \neg(\mathsf{Close} = \mathsf{Forward}) \wedge \neg(\mathsf{Close} = \mathsf{Backward}) \wedge \neg(\mathsf{Close} = \mathsf{Neutral}) \wedge$
$\neg(\mathsf{Forward} = \mathsf{Backward}) \wedge \neg(\mathsf{Forward} = \mathsf{Neutral}) \wedge \neg(\mathsf{Backward} = \mathsf{Neutral})$

Dir_Induct $\vdash \forall P.\, P\ \mathsf{Close} \wedge P\ \mathsf{Forward} \wedge P\ \mathsf{Backward} \wedge P\ \mathsf{Neutral} \supset (\forall D.\, P\ D)$

Dir_Cases $\vdash \forall D.\,(D = \mathsf{Close}) \vee (D = \mathsf{Forward}) \vee (D = \mathsf{Backward}) \vee (D = \mathsf{Neutral})$

Zero_Eq_0 $\vdash \forall t.\, \mathsf{Zero\ Eq}\ 0\ t$

ArbNot0_Not_Zero $\vdash \neg(\mathsf{ArbNot0}\ t = 0)$

Arb_True $\vdash \forall a\, b.\, a \vee a \vee \neg a \vee b$

Mode_dist1 $\vdash \forall M.\,(M = \mathsf{Reset}) \vee (M = \mathsf{Idle}) \vee (M = \mathsf{Shutdown}) \vee (M = \mathsf{Traction}) \vee$
$(M = \mathsf{Manual}) \supset \neg(M = \mathsf{Cruise})$

Mode_dist2 $\vdash \neg(\mathsf{Cruise} = \mathsf{Shutdown})$

Mode_dist3 $\vdash \neg(\mathsf{Manual} = \mathsf{Shutdown})$

Dir_lemma $\vdash \forall t.\, \mathsf{Always}$
$(\mathsf{DecodeCtrl}\ (pwm\_act, forward, backward, neutral, close)$
$\longrightarrow \mathsf{Xor2}\ (close, \mathsf{Xor3}\ (forward, backward, neutral)))\ t$

STATE_EQ1 $\vdash \forall s\, s'\, t.\,(s = s') \wedge \neg\mathsf{RUNNING}\ s\ t \supset \neg\mathsf{RUNNING}\ s'\ t$

STATE2_EQ $\vdash \forall s\, s'\, t.\,(s = s') \wedge \neg\mathsf{WFAILED}\ s\ t \supset \neg\mathsf{WFAILED}\ s'\ t$

STOP_SAFE $\vdash \forall s\, m\, v\, p\, t.\, \mathsf{Always}\ (\mathsf{Not}\ (\mathsf{RUNNING}\ s) \longrightarrow \mathsf{SAFE}\ (s, m, v, p))\ t$

FAIL_SAFE $\vdash \forall s\, m\, v\, p\, t.$
$\mathsf{Always}\ ((\mathsf{WFAILED}\ s\ \mathsf{And}\ \mathsf{RUNNING}\ s) \longrightarrow \mathsf{SAFE}\ (s, m, v, p))\ t$

INIT_SAFE $\vdash \forall s\, m\, v\, p\, t.\, \mathsf{Always}\ (\mathsf{INIT}\ (s, m, v, p) \longrightarrow \mathsf{SAFE}\ (s, m, v, p))\ t$

STATES_UNIQUE $\vdash \forall m\ v\ p\ t. (\exists s. (\exists \forall m'\ v'\ p'. \text{NextState}\ (s, m, v, p)\ t = m', v', p'))$

SAFETY $\vdash \forall t. (\text{INIT}\ (s\ t, m\ t, v\ t, p\ t)\ t \supset \text{SAFE}\ (s\ t, m\ t, v\ t, p\ t)\ t)\ \wedge$

$(\text{BACK\_BEHAV}\ (s, m, v, p)\ t\ \wedge\ \text{SAFE}\ (s\ t, m\ t, v\ t, p\ t)\ t$

$\supset \text{State\_Safe}\ (s\ t, m\ (t+1), v\ (t+1), p\ (t+1))\ t)$

RCOND_lemma $\vdash \forall t\ s\ m\ v\ p.$

$(s = \text{False}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb})$

$\supset ((m = \text{Cruise}) \Rightarrow (\text{NextState}\ (s, m, v, p)\ t = \text{Reset}, \text{F}, \text{Close})\ |$

$(\text{NextState}\ (s, m, v, p)\ t = \text{Reset}, v, \text{Close}))$

SCOND1_lemma $\vdash \forall t\ s\ m\ v\ p. (m = \text{Shutdown})\ \wedge$

$(s = \text{True}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb})$

$\supset (\text{NextState}\ (s, m, v, p)\ t = \text{Shutdown}, v, \text{Close})$

SCOND2_lemma $\vdash \forall t\ s\ m\ v\ p.$

$(s = \text{True}, \text{Arb}, \text{Arb}, \text{True}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb})\ \vee$

$(s = \text{True}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{True}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb})\ \vee$

$(s = \text{True}, \text{Arb}, \text{Arb}, \text{Arb}, \text{True}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{ArbNot0}, \text{Arb}, \text{Arb})$

$\supset ((m = \text{Cruise}) \Rightarrow (\text{NextState}\ (s, m, v, p)\ t = \text{Shutdown}, \text{T}, \text{Close})\ |$

$(\text{NextState}\ (s, m, v, p)\ t = \text{Shutdown}, v, \text{Close}))$

SCOND3_lemma $\vdash \forall t\ s\ m\ v\ p.$

$(s = \text{True}, \text{Arb}, \text{Arb}, \text{Arb}, \text{False}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Zero}, \text{Arb}, \text{Arb})$

$\supset ((m = \text{Cruise}) \Rightarrow (\text{NextState}\ (s, m, v, p)\ t = \text{Shutdown}, \text{T}, \text{Close})\ |$

$(\text{NextState}\ (s, m, v, p)\ t = \text{Shutdown}, v, \text{Close}))$

SCOND4_lemma $\vdash \forall t\ s\ m\ v\ p.$

$(s = \text{True}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{True}, \text{Arb}, \text{Arb}, \text{Arb}, \text{ArbNot0}, \text{Arb}, \text{Arb})\ \vee$

$(s = \text{True}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Arb}, \text{False}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Zero}, \text{Arb}, \text{Arb})$

$\supset ((m = \text{Cruise}) \Rightarrow (\text{NextState}\ (s, m, v, p)\ t = \text{Shutdown}, \text{T}, \text{Close})\ |$

$(\text{NextState}\ (s, m, v, p)\ t = \text{Shutdown}, v, \text{Close}))$

ICOND1_lemma $\vdash \forall t\ s\ m\ v\ p. (m = \text{Cruise})\ \wedge$

$(s = \text{True}, \text{False}, \text{Arb}, \text{False}, \text{False}, \text{False}, \text{True}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Zero}, \text{Arb}, \text{Arb})$

$\supset (\text{NextState}\ (s, m, v, p)\ t = \text{Idle}, \text{T}, \text{PWM}\ (p, \text{FBANG}\ s\ t, 1))$

ICOND2_lemma $\vdash \forall t\ s\ m\ v\ p. (m = \text{Cruise})\ \wedge$

$(s = \text{True}, \text{Arb}, \text{True}, \text{False}, \text{False}, \text{False}, \text{True}, \text{Arb}, \text{Arb}, \text{Arb}, \text{Zero}, \text{Arb}, \text{Arb})$

$\supset (\text{NextState}\ (s, m, v, p)\ t = \text{Idle}, \text{T}, \text{PWM}\ (p, \text{FBANG}\ s\ t, 1))$

ICOND3_lemma $\vdash \forall t\ s\ m\ v\ p. (m = \text{Manual}) \wedge$

$((s = \text{True, False, False, False, False, False, True, Arb, Arb, Arb, Zero, Arb, Arb}) \vee$

$(s = \text{True, False, True, False, False, False, True, Arb, Arb, Arb, Zero, Arb, Arb}) \vee$

$(s = \text{True, True, True, False, False, False, True, Arb, Arb, Arb, Zero, Arb, Arb}))$

$\supset (\text{NextState}\ (s, m, v, p)\ t = \text{Idle}, v, \text{PWM}\ (p, \text{FBANG}\ s\ t, 1))$

CCOND1_lemma $\vdash \forall t\ s\ m\ v\ p. (m = \text{Manual}) \wedge$

$(s = \text{True, True, False, False, False, False, False, Arb, True,}$

$\text{Arb, ArbNot0, ArbNot0, Arb})$

$\supset (\text{NextState}\ (s, m, v, p)\ t = \text{Cruise}, v, \text{PWM}\ (p, \text{FBANG}\ s\ t, \text{SPEED}\ s\ t))$

CCOND2_lemma $\vdash \forall t\ s\ m\ v\ p. (m = \text{Idle}) \wedge$

$(s = \text{True, True, False, False, False, False, True, Arb, Arb, True, Zero, Arb, Arb})$

$\supset (\text{NextState}\ (s, m, \text{T}, p)\ t = \text{Cruise}, \text{T}, \text{PWM}\ (p, \text{FBANG}\ s\ t, \text{SPEED}\ s\ t))$

TCOND_lemma $\vdash \forall t\ s\ m\ v\ p. (m = \text{Manual}) \wedge$

$(s = \text{True, False, Arb, False, False, False, False, True, Arb, Arb, ArbNot0, Zero, Arb})$

$\supset (\text{NextState}\ (s, m, v, p)\ t = \text{Traction}, v, \text{PWM}\ (p, \text{FBANG}\ s\ t, \text{DEMAND}\ s\ t))$

MCOND_lemma $\vdash \forall t\ s\ m\ v\ p. \neg(m = \text{Shutdown}) \wedge$

$(s = \text{True, Arb, Arb, False, False, False, False, False, False, False, ArbNot0, Arb, Arb})$

$\supset (\text{NextState}\ (s, m, v, p)\ t = \text{Manual}, \text{F}, \text{PWM}\ (p, \text{FBANG}\ s\ t, \text{DEMAND}\ s\ t))$

FSM $\vdash (\forall ip\ op. \text{Init}\ (ip, op) \supset \text{Safe}\ (ip, op)) \wedge$

$(\forall ip\ op\ ip'\ op'. \text{Nextstate}\ (ip, op)\ (ip', op') \wedge \text{Safe}\ (ip, op) \supset \text{Safe}\ (ip', op'))$

$\supset (\forall e. \text{LSA}\ (\text{Init, Nextstate})\ e = \text{PLSA}\ (\text{Init, Safe, Nextstate})\ e)$

C.3 Procedures of Proofs and Tactics

```
let Mode_Axiom = define_type 'Mode_Axiom'
  'Mode=Reset|Idle|Traction|Shutdown|Manual|Cruise';;


let Mode_Distinct = save_thm ('Mode_Distinct',
  prove_constructors_distinct Mode_Axiom);;


let Mode_Induct = save_thm ('Mode_Induct',
  prove_induction_thm Mode_Axiom);;
```

```
let Mode_Cases = save_thm (`Mode_Cases`,
  prove_cases_thm Mode_Induct);;


let sig_ty = ":(num->bool)#(num->bool)#(num->bool)#
               (num->bool)#(num->bool)#(num->bool)#
               (num->bool)#(num->bool)#(num->bool)#
               (num->bool)#(num->num)#(num->num)#
               (num->num)";;


new_type_abbrev (`Cntrl_ty`, ":bool#bool#bool#bool#bool#
                               bool#bool#bool#bool#bool#
                               num#num#num");;


let RUNNING = new_definition (`RUNNING_DEF`,
  "RUNNING (s:^sig_ty) = FST s
  ");;
let GEAR = new_definition (`GEAR_DEF`,
  "GEAR (s:^sig_ty)=FST(SND s)
  ");;
let BRAKE = new_definition (`BRAKE_DEF`,
  "BRAKE (s:^sig_ty)=FST(SND(SND s)
  )");;
let WATCHOUT = new_definition (`WATCHOUT_DEF`,
  "WATCHOUT (s:^sig_ty)=FST(SND(SND(SND s)
  ))");;
let WFAILED = new_definition (`WFAILED_DEF`,
  "WFAILED (s:^sig_ty)=FST(SND(SND(SND(SND s)
  )))");;
let OVERTEMP = new_definition (`OVERTEMP_DEF`,
  "OVERTEMP (s:^sig_ty)=FST(SND(SND(SND(SND(SND s)
   ))))");;
let IDLE = new_definition (`IDLE_DEF`,
  "IDLE (s:^sig_ty)=FST(SND(SND(SND(SND(SND(SND s)
   )))))");;
```

```
let TRACTION = new_definition ('TRACTION_DEF',
  "TRACTION (s:^sig_ty)=FST(SND(SND(SND(SND(SND(SND(SND s)
   ))))))");;
let CRUISE = new_definition ('CRUISE_DEF',
  "CRUISE (s:^sig_ty)=FST(SND(SND(SND(SND(SND(SND(SND(SND s)
   )))))))");;
let RESUME = new_definition ('RESUME_DEF',
  "RESUME (s:^sig_ty)=FST(SND(SND(SND(SND(
          SND(SND(SND(SND(SND s)))))))))");;
let DEMAND = new_definition ('DEMAND_DEF',
  "DEMAND (s:^sig_ty)= FST(SND(SND(SND(SND(SND(SND
          (SND(SND(SND(SND s))))))))))");;
let SPEED = new_definition ('SPEED_DEF',
  "SPEED (s:^sig_ty)= FST(SND(SND(SND(SND(SND(SND(SND
          (SND(SND(SND(SND s)))))))))))");;
let ANGLE = new_definition ('ANGLE_DEF',
  "ANGLE (s:^sig_ty)= SND(SND(SND(SND(SND(SND(SND(SND
           (SND(SND(SND(SND s)))))))))))");;


%-------------------------------------------------------------
% Reset state %
%-------------------------------------------------------------
let R_COND = new_definition ('R_COND_DEF',
  "R_COND (running)=
    \(t:time). Not running t");;


%-------------------------------------------------------------
% Idle state %
%-------------------------------------------------------------
let I_COND = new_definition ('I_COND_DEF',
  "I_COND (mode,idle,brake,gear)=
    \(t:time).
     ((mode=Manual) \/ (mode=Cruise) \/ (mode=Idle)) /\
         idle t /\ (brake t \/ ~gear t)");;
```

```
%------------------------------------------------------------
% Shutdown state %
%------------------------------------------------------------
let S_COND = new_definition (`S_COND_DEF`,
  "S_COND (watchout,overtemp,stop)=
    \(t:time).
      watchout t \/ overtemp t \/ stop");;

%------------------------------------------------------------
% Cruise state %
%------------------------------------------------------------
let C_COND = new_definition (`C_COND_DEF`,
  "C_COND (cruise,speed,resume,valid_cruise,
          gear,mode,demand,brake) = \(t:time).
    ((cruise t /\ ~(speed Eq 0)t /\ (mode=Manual)) \/
      (~(speed t < demand t) /\ ~brake t /\ (mode=Cruise))
        \/ (resume t /\ valid_cruise /\ (mode=Idle)))
         /\ gear t");;

%------------------------------------------------------------
% Traction state %
%------------------------------------------------------------
let T_COND = new_definition (`T_COND_DEF`,
  "T_COND (mode,traction,speed,gear,idle)= \(t:time).
     ((mode=Manual) \/ (mode=Traction)) /\
       traction t /\ (speed Eq 0)t /\ ~(gear t)
         /\ ~(idle t)");;

%------------------------------------------------------------
%  Movement of actuator's throttle
%------------------------------------------------------------
let Dir_Axiom = define_type `Dir_Axiom`
  `Dir = Close | Forward | Backward | Neutral`;;
```

```
let Dir_Distinct = save_thm (`Dir_Distinct`,
  prove_constructors_distinct Dir_Axiom);;

let Dir_Induct = save_thm (`Dir_Induct`,
  prove_induction_thm Dir_Axiom);;

let Dir_Cases = save_thm (`Dir_Cases`,
  prove_cases_thm Dir_Induct);;

%-----------------------------------------------------------
% Actuator behavioural spec.
%-----------------------------------------------------------
let PWM = new_definition (`PWM_DEF`,
  "PWM (pwm_act,ang,dem) =
    (dem=0)   => Close |
    (dem>ang) => ((pwm_act = Forward)  =>
                 Neutral|Forward) |
    (ang>dem) => ((pwm_act = Backward) =>
                 Neutral|Backward) | Neutral");;

%-----------------------------------------------------------
% Validity conditions of interface signals
%-----------------------------------------------------------
let ILLEGAL = new_definition(`ILLEGAL_DEF`,
  "ILLEGAL b p = \(t:time). ~Xor2(p,b)t");;

let DecodeSig = new_definition  (`DecodeSig_DEF`,
  "DecodeSig ((sig:time->Cntrl_ty),
    (runningt:time->bool),(geart:time->bool),
    (braket:time->bool),(watchoutt:time->bool),
    (wfailedt:time->bool),(overtempt:time->bool),
    (idlet:time->bool),(tractiont:time->bool),
    (cruiset:time->bool),(resumet:time->bool),
    (demandt:time->num),(speedt:time->num),
    (anglet:time->num)) =
     !(t:time).((runningt t),(geart t),(braket t),
                (watchoutt t),(wfailedt t),(overtempt t),
                (idlet t),(tractiont t),(cruiset t),
                (resumet t),(demandt t),(speedt t),
                (anglet t)) = (sig t)");;

%------------------------------------------------------------
% Main loop %
%------------------------------------------------------------
let NextState = new_definition ('NextState_DEF',
  "NextState(s,mode,valid_cruise,pwm_act) =
    \(t:time).
    (let running = RUNNING s in
    (let gear = GEAR s in
    (let brake = BRAKE s in
    (let watchout = WATCHOUT s in
    (let wfailed = WFAILED s in
    (let overtemp = OVERTEMP s in
    (let idle = IDLE s in
    (let traction = TRACTION s in
    (let cruise = CRUISE s in
    (let resume = RESUME s in
    (let demand = DEMAND s in
    (let speed = SPEED s in
    (let angle = ANGLE s in
    (let stop_ok = (ILLEGAL idle (Not(demand Eq 0)))t
         \/ (wfailed t) \/ (mode=Shutdown) in
    (let reset_ok = R_COND(running)t in
    (let cruise_ok = ~reset_ok /\
           C_COND(cruise,speed,resume,valid_cruise,
                  gear,mode,demand,brake)t in
    (let traction_ok = ~reset_ok /\
          T_COND(mode,traction,speed,gear,idle)t in
    (let shutdown_ok = S_COND(watchout,overtemp,stop_ok)t in
    (let idle_ok = ~reset_ok /\
          I_COND(mode,idle,brake,gear)t in
    (let valid_c = ((mode=Cruise) => ~reset_ok |
                valid_cruise) in
    reset_ok=>
        (Reset,valid_c,PWM(pwm_act,angle t,~zerodem))|
    shutdown_ok=>
        (Shutdown,valid_c,PWM(pwm_act,angle t,~zerodem))|
    idle_ok=>
        (Idle,valid_c,PWM(pwm_act,angle t,~idledem))|
    cruise_ok=>
        (Cruise,valid_c,PWM(pwm_act,angle t,speed t))|
    traction_ok=>
        (Traction,valid_c,PWM(pwm_act,angle t,demand t))|
        (Manual,F,PWM(pwm_act,angle t,demand t))
    ))))))))))))))))))))");;


let BACK_BEHAV = new_definition ('BACK_BEHAV_DEF',
  "BACK_BEHAV ((sigt:time->^sig_ty),(modet:time->Mode),
            (valid_ct:time->bool),(pwm_actt:time->Dir)) =
    \(t:time).
       NextState ((sigt t), modet t, valid_ct t, pwm_actt t)t
          = (modet(t+1), valid_ct(t+1), pwm_actt(t+1))");;


let DecodeCtrl = new_definition ('DecodeCrtl',
  "DecodeCtrl (pwm_act,forward,backward,neutral,close) =
    \(t:time). (forward t = (pwm_act = Forward)) /\
               (backward t = (pwm_act = Backward)) /\
               (neutral t = (pwm_act = Neutral)) /\
               (close t = (pwm_act = Close))");;


let Accelerate = new_definition ('Accelerate_DEF',
  "Accelerate (act_start,forward) = act_start And forward");;


let Decelerate = new_definition (`Decelerate_DEF`,
  "Decelerate (act_start,backward) =
        act_start And backward");;


let RESPONSE = new_definition (`RESPONSE_DEF`,
  "RESPONSE (forward,backward,neutral,output,act_start) =
    \(t:time).
      forward t =>
       ((output Within (Len (act_start) ^Tres))
                  When act_start)|
      backward t =>
       ((output Within (Len (act_start) ^Tres))
                  When act_start)|
      neutral t");;


let REGISTER = new_definition (`REGISTER_DEF`,
  "REGISTER (act_stop,act_start,neutral) =
     \(t:time).
       Always ((act_stop And (Not neutral)) -->
        (act_start Within (Len (Not neutral) ^Treg)))t");;


let Below = new_infix_definition (`Below_DEF`,
  "Below pos ang = \t. (pos Eq (PRE ang))t");;


let Above = new_infix_definition (`Above_DEF`,
  "Above pos ang = \t. (pos Eq (SUC ang))t");;


let TRANSIT = new_definition (`TRANSIT_DEF`,
  "TRANSIT (accelerating,decelerating) =
    \(t:time). !pos ang.
     Always (
      ((accelerating And Not(pos Below ang)) -->
      (Not(pos Below ang) Within
           (Len accelerating ^Ttr))) And
     ((decelerating And Not(pos Above ang)) -->
      (Not(pos Above ang) Within
           (Len decelerating ^Ttr))))t");;


let POS_DIR = new_definition (`POS_DIR_DEF`,
  "POS_DIR (act_stop,accelerating,decelerating,
           close,forward,backward,neutral) =
    \(t:time).
      act_stop t     =>  Next(close)t |
      accelerating t => (forward t => Next(forward)t |
                         backward t => Next(Not forward)t |
                         Next(neutral)t) |
      decelerating t => (forward t => Next(Not forward)t |
                         backward t => Next(backward)t |
                         Next(neutral)t) |
      Next(neutral)t");;


let PERFORM1 = new_definition (`PERFORM1_DEF`,
  "PERFORM1 (act_stop,act_start,accelerating,decelerating,
            forward,backward,neutral,close) =
    \(t:time).
      Always (act_stop -->
        Next (neutral t => act_stop |
             (act_start And Not decelerating)t => forward |
             (act_start And Not accelerating)t => backward |
              close))t");;


let PERFORM2 = new_definition (`PERFORM2_DEF`,
  "PERFORM2 (act_start,neutral) =
     \(t:time). Always (act_start --> Not neutral)t");;


let PERFORM3 = new_definition (`PERFORM3_DEF`,
  "PERFORM3 (forward,backward,neutral,act_stop,act_start) =
  \(t:time).
     Always (
      forward t => Next(Not(forward)t =>
                        act_stop | forward)|
      backward t=> Next(Not(backward)t =>
                        act_stop | backward)|
      neutral t => Next(Not(neutral)t =>
                        act_start | act_stop)|
      act_stop)t");;

let PERFORM4 = new_definition ('PERFORM4_DEF',
  "PERFORM4 (forward,backward,neutral,
             accelerating,decelerating) =
    \(t:time).
       Always (
         forward t  => Next(accelerating t =>
                            forward | neutral)|
         backward t => Next(decelerating t =>
                            backward | neutral)|
         Next(neutral))t");;

let FORE_BEHAV = new_definition ('FORE_BEHAV_DEF',
  "FORE_BEHAV (pwm_act,output) =
    \(t:time). !act_start act_stop close forward
               backward neutral.
       (let accelerating = Accelerate(act_start,forward) in
       (let decelerating = Decelerate(act_start,backward) in
       (DecodeCtrl (pwm_act,forward,backward,neutral,close)
        And
        RESPONSE (forward,backward,neutral,output,act_start)
        And
        REGISTER (act_stop,act_start,neutral) And
        TRANSIT (accelerating,decelerating) And
        POS_DIR (act_stop,accelerating,decelerating,
                 close,forward,backward,neutral) And
        PERFORM1 (act_stop,act_start,accelerating,
          decelerating,forward,backward,neutral,close) And
        PERFORM2 (act_start,neutral) And
        PERFORM3 (forward,backward,neutral,
                  act_stop,act_start) And
        PERFORM4 (forward,backward,neutral,
                  accelerating,decelerating))t))");;


let DBW_BEHAV = new_definition (`DBW_BEHAV_DEF`,
  "DBW_BEHAV =
    !(t:time) sigt modet valid_ct pwm_actt output timer.
     Always ((BACK_BEHAV(sigt,modet,valid_ct,pwm_actt)
             Upon timer) (FORE_BEHAV(pwm_actt t,output)))t
          /\ Valid timer");;


let State_Safe = new_definition (`State_Safe_DEF`,
  "State_Safe (s,(m:Mode),(v:bool),p) =
      \t. ~(WFAILED s)t \/ (p=Close)");;


let SAFE = new_definition (`SAFE_DEF`,
  "SAFE (s,m,v,p) =
        \t. State_Safe(s,(NextState (s,m,v,p)t))t");;


let INIT = new_definition (`INIT_DEF`,
  "INIT (s,(m:Mode),(v:bool),(p:Dir)) =
      \t. ~(RUNNING s)t /\ (DEMAND s Eq 0)t");;


let ArbNot0 = new_definition (`ArbNot0_DEF`,
  "ArbNot0 = \(t:time). @n. ~(n=0)");;


let Arb = new_definition (`Arb_DEF`,
  "Arb = \(t:time). (ARB:*)");;
```

```
let Zero = new_definition ('Zero_DEF',
  "Zero = \(t:time). 0");;

let Zero_Eq_0 = prove_thm ('Zero_Eq_0',
  "!t. (Zero Eq 0)t",
    GEN_TAC THEN REWRITE_TAC[Zero;Eq_DEF]);;

let ArbNot0_Not_Zero = prove_thm('ArbNot0_Not_Zero',
  "~(ArbNot0 t=0)",
   REWRITE_TAC[ArbNot0] THEN CONV_TAC SELECT_CONV
   THEN EXISTS_TAC "SUC 0"
   THEN MATCH_ACCEPT_TAC SUC_ID);;

let Arb_True = prove_thm ('Arb_True',
  "!a b. a \/ a \/ ~a \/ b",
   REPEAT GEN_TAC THEN BOOL_CASES_TAC "a"
   THEN REWRITE_TAC[]);;

let Mode_dist1 = prove_thm('Mode_dist1',
  "!M. (M=Reset)\/(M=Idle)\/(M=Shutdown)\/
       (M=Traction)\/(M=Manual) ==> ~(M=Cruise)",
   GEN_TAC THEN STRIP_TAC THEN ASM_REWRITE_TAC[]
   THEN REWRITE_TAC[Mode_Distinct]);;

let Mode_dist2 = prove_thm('Mode_dist2',
  "~(Cruise=Shutdown)",
   MP_TAC Mode_Distinct THEN STRIP_TAC
   THEN POP_ASSUM (\th1. POP_ASSUM (\th2. (ASSUME_TAC th2)))
   THEN CONV_TAC (ONCE_DEPTH_CONV SYM_CONV)
   THEN ASM_REWRITE_TAC[]);;

let Mode_dist3 = prove_thm('Mode_dist3',
  "~(Manual=Shutdown)",
   MP_TAC Mode_Distinct THEN STRIP_TAC
   THEN POP_ASSUM (\th1. POP_ASSUM (\th2. (ASSUME_TAC th2)))
   THEN CONV_TAC (ONCE_DEPTH_CONV SYM_CONV)
   THEN ASM_REWRITE_TAC[]);;


let Dir_lemma = prove_thm ('Dir_lemma',
  "!(t:time). Always (
       DecodeCtrl (pwm_act,forward,backward,neutral,close)
         --> Xor2(close,Xor3(forward,backward,neutral)))t",
     GEN_TAC
     THEN REWRITE_TAC[Xor2_DEF;Xor3_DEF;Always_DEF;
                      DecodeCtrl;Imp_DEF]
     THEN BETA_TAC THEN REPEAT STRIP_TAC
     THENL [%1%
            ASM_REWRITE_TAC[] THEN ASSUME_TAC (Dir_Cases)
            THEN POP_ASSUM MP_TAC
            THEN CONV_TAC LEFT_IMP_FORALL_CONV
            THEN EXISTS_TAC "pwm_act" THEN REPEAT STRIP_TAC
            THENL [ASM_REWRITE_TAC[];
                   ASM_REWRITE_TAC[Dir_Distinct];
                   ASM_REWRITE_TAC[]
                   THEN CONV_TAC(ONCE_DEPTH_CONV SYM_CONV)
                   THEN REWRITE_TAC[Dir_Distinct]
                   THEN CONV_TAC(ONCE_DEPTH_CONV SYM_CONV)
                   THEN REWRITE_TAC[Dir_Distinct];
                   ASM_REWRITE_TAC[]
                   THEN CONV_TAC(ONCE_DEPTH_CONV SYM_CONV)
                   THEN REWRITE_TAC[Dir_Distinct]];
            POP_ASSUM (\th. ALL_TAC) THEN RES_TAC
            THEN POP_ASSUM MP_TAC THEN ASM_REWRITE_TAC[]
            THEN REWRITE_TAC[Dir_Distinct];
            POP_ASSUM (\th. ALL_TAC) THEN RES_TAC
            THEN POP_ASSUM MP_TAC THEN ASM_REWRITE_TAC[]
            THEN REWRITE_TAC[Dir_Distinct];
            POP_ASSUM (\th. ALL_TAC) THEN RES_TAC
            THEN POP_ASSUM MP_TAC THEN ASM_REWRITE_TAC[]
            THEN REWRITE_TAC[Dir_Distinct]]);;

let STATE_EQ1 = prove_thm ('STATE_EQ1',
  "!s s' t. (s=s') /\ ~(RUNNING s)t ==> ~(RUNNING s')t",
    REPEAT GEN_TAC THEN STRIP_TAC
    THEN POP_ASSUM MP_TAC THEN ASM_REWRITE_TAC[]);;

let STATE_EQ2 = prove_thm ('STATE2_EQ',
  "!s s' t. (s=s') /\ ~(WFAILED s)t ==> ~(WFAILED s')t",
    REPEAT GEN_TAC THEN STRIP_TAC
    THEN POP_ASSUM MP_TAC THEN ASM_REWRITE_TAC[]);;

%-------------------------------------------------------------
%                  Proving Safety Properties
%-------------------------------------------------------------
let STOP_SAFE = prove_thm ('STOP_SAFE',
  "!s m v p t. Always (Not(RUNNING s) --> SAFE(s,m,v,p))t",
     REPEAT GEN_TAC
     THEN REWRITE_TAC[SAFE;Always_DEF;Imp_DEF]
     THEN REWRITE_TAC[NextState;S_COND;R_COND;Not_DEF]
     THEN BETA_TAC THEN REPEAT STRIP_TAC
     THEN CONV_TAC(DEPTH_CONV let_CONV)
     THEN ASM_REWRITE_TAC[]
     THEN REWRITE_TAC[State_Safe]
     THEN BETA_TAC THEN REWRITE_TAC[PWM]);;

let FAIL_SAFE = prove_thm ('FAIL_SAFE',
  "!s m v p t.
    Always((WFAILED s And RUNNING s)-->SAFE(s,m,v,p))t",
     REPEAT GEN_TAC
     THEN REWRITE_TAC[SAFE;Always_DEF;Imp_DEF]
     THEN REWRITE_TAC[NextState;S_COND;
```

```
        R_COND;And_DEF;Not_DEF]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN CONV_TAC(DEPTH_CONV let_CONV)
    THEN ASM_REWRITE_TAC[IMP_CLAUSES]
    THEN REWRITE_TAC[State_Safe]
    THEN BETA_TAC THEN REWRITE_TAC[PWM]);;

let INIT_SAFE = prove_thm (`INIT_SAFE`,
  "!s m v p t. Always(INIT(s,m,v,p) --> SAFE(s,m,v,p))t",
    REWRITE_TAC[Always_DEF;Imp_DEF;INIT;SAFE]
    THEN REWRITE_TAC[NextState;R_COND;Not_DEF]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN CONV_TAC(DEPTH_CONV let_CONV)
    THEN ASM_REWRITE_TAC[IMP_CLAUSES]
    THEN REWRITE_TAC[PWM;State_Safe]);;

let STATES_UNIQUE = prove_thm (`STATES_UNIQUE`,
  "!m v p t. ?s. ?!m' v' p'.
        NextState(s,m,v,p)t = (m',v',p')",
    REPEAT GEN_TAC
    THEN EXISTS_TAC "False,(Arb:num->bool),(Arb:num->bool),
                     (Arb:num->bool),(Arb:num->bool),
                     (Arb:num->bool),(Arb:num->bool),
                     (Arb:num->bool),(Arb:num->bool),
                     (Arb:num->bool),(Arb:num->num),
                     (Arb:num->num),(Arb:num->num)"
    THEN PURE_REWRITE_TAC[False_DEF;NextState;RUNNING;FST]
    THEN CONV_TAC(DEPTH_CONV let_CONV)
    THEN PURE_REWRITE_TAC[R_COND;Not_DEF] THEN BETA_TAC
    THEN REWRITE_TAC[NOT_CLAUSES;ANGLE;SND;PWM]
    THEN CONV_TAC EXISTS_UNIQUE_CONV THEN STRIP_TAC
    THENL [EXISTS_TAC "Reset"
       THEN CONV_TAC EXISTS_UNIQUE_CONV THEN STRIP_TAC
       THENL [EXISTS_TAC "(m=Cruise)=>F|v"
```

```
           THEN CONV_TAC EXISTS_UNIQUE_CONV THEN STRIP_TAC
           THENL [EXISTS_TAC "Close" THEN REFL_TAC;
                  REPEAT STRIP_TAC THEN POP_ASSUM MP_TAC
                  THEN PURE_ASM_REWRITE_TAC[]
                  THEN PURE_REWRITE_TAC[PAIR_EQ]
                  THEN REPEAT STRIP_TAC];
           REPEAT GEN_TAC
           THEN CONV_TAC(DEPTH_CONV EXISTS_UNIQUE_CONV)
           THEN STRIP_TAC THEN POP_ASSUM (\t. ALL_TAC)
           THEN POP_ASSUM MP_TAC THEN PURE_ASM_REWRITE_TAC[]
           THEN PURE_REWRITE_TAC[PAIR_EQ]
           THEN REPEAT STRIP_TAC];
        REPEAT GEN_TAC
        THEN CONV_TAC(DEPTH_CONV EXISTS_UNIQUE_CONV)
        THEN STRIP_TAC THEN POP_ASSUM (\th. ALL_TAC)
        THEN POP_ASSUM (\th. ALL_TAC) THEN POP_ASSUM MP_TAC
        THEN PURE_ASM_REWRITE_TAC[]
        THEN PURE_REWRITE_TAC[PAIR_EQ]
        THEN REPEAT STRIP_TAC]);;


let SAFETY = prove_thm ('SAFETY',
  "!t. (INIT(s t,m t,v t,p t)t ==> SAFE(s t,m t,v t,p t)t) /\
       ((BACK_BEHAV(s,m,v,p)t /\ SAFE(s t,m t,v t,p t)t)
          ==> State_Safe(s t,m(t+1),v(t+1),p(t+1))t)",
      GEN_TAC THEN STRIP_TAC
      THENL [PURE_REWRITE_TAC[INIT;SAFE]
             THEN BETA_TAC THEN REPEAT STRIP_TAC
             THEN PURE_REWRITE_TAC[NextState;R_COND;Not_DEF]
             THEN BETA_TAC THEN CONV_TAC(DEPTH_CONV let_CONV)
             THEN ASM_REWRITE_TAC[PWM]
             THEN PURE_REWRITE_TAC[State_Safe]
             THEN BETA_TAC THEN REWRITE_TAC[];
             PURE_REWRITE_TAC[BACK_BEHAV;SAFE] THEN BETA_TAC
             THEN REPEAT STRIP_TAC THEN POP_ASSUM MP_TAC
```

```
            THEN ASM_REWRITE_TAC[]]);;


%------------------------------------------------------------
%              Proving state transition diagrams
%------------------------------------------------------------
% Reset transition condition %
let RCOND_lemma = prove_thm(`RCOND_lemma`,
  "!(t:time) s m v p.
       (s=False,Arb,Arb,Arb,Arb,Arb,Arb,
               Arb,Arb,Arb,Arb,Arb,Arb)
      ==> (m=Cruise)=>(NextState(s,m,v,p)t=(Reset,F,Close))|
                     (NextState(s,m,v,p)t=(Reset,v,Close))",
     REPEAT GEN_TAC THEN PURE_REWRITE_TAC[NextState]
     THEN DISCH_THEN (\th. PURE_REWRITE_TAC[th])
     THEN PURE_REWRITE_TAC[RUNNING;R_COND]
     THEN CONV_TAC (DEPTH_CONV let_CONV)
     THEN PURE_REWRITE_TAC[False_DEF;Not_DEF]
     THEN BETA_TAC THEN COND_CASES_TAC
     THEN REWRITE_TAC[PWM]);;


% Shutdown transition condition 1 %
let SCOND1_lemma = prove_thm(`SCOND1_lemma`,
  "!(t:time) s m v p. (m=Shutdown) /\
       (s=True,Arb,Arb,Arb,Arb,Arb,Arb,
            Arb,Arb,Arb,Arb,Arb,Arb)
      ==> (NextState(s,m,v,p)t=(Shutdown,v,Close))",
    REPEAT GEN_TAC
    THEN PURE_REWRITE_TAC[NextState;R_COND;RUNNING;S_COND;
       WATCHOUT;OVERTEMP;WFAILED;DEMAND;IDLE;ILLEGAL;
       Xor2_DEF;False_DEF;True_DEF;Not_DEF]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[]
    THEN CONV_TAC (DEPTH_CONV let_CONV)
    THEN REWRITE_TAC[PWM;Mode_Distinct]);;
```

```
% Shutdown transition condition 2 %
let SCOND2_lemma = prove_thm(`SCOND2_lemma`,
  "!(t:time) s m v p.
    (s=True,Arb,Arb,True,Arb,Arb,Arb,
          Arb,Arb,Arb,Arb,Arb,Arb) \/
    (s=True,Arb,Arb,Arb,Arb,True,Arb,
          Arb,Arb,Arb,Arb,Arb,Arb) \/
    (s=True,Arb,Arb,Arb,True,Arb,Arb,
          Arb,Arb,Arb,ArbNot0,Arb,Arb) ==>
    (m=Cruise)=>(NextState(s,m,v,p)t=(Shutdown,T,Close))|
                (NextState(s,m,v,p)t=(Shutdown,v,Close))",
    REPEAT GEN_TAC
    THEN PURE_REWRITE_TAC[NextState;R_COND;RUNNING;S_COND;
       WATCHOUT;OVERTEMP;WFAILED;DEMAND;IDLE;ILLEGAL;
       Xor2_DEF;True_DEF;False_DEF;Not_DEF]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[]
    THEN CONV_TAC (DEPTH_CONV let_CONV)
    THEN REWRITE_TAC[] THEN COND_CASES_TAC
    THEN REWRITE_TAC[PWM]);;

% Shutdown transition condition 3 %
let SCOND3_lemma = prove_thm(`SCOND3_lemma`,
  "!(t:time) s m v p.
     (s=True,Arb,Arb,Arb,False,Arb,
         Arb,Arb,Arb,Arb,Zero,Arb,Arb) ==>
     (m=Cruise)=>(NextState(s,m,v,p)t=(Shutdown,T,Close))|
                 (NextState(s,m,v,p)t=(Shutdown,v,Close))",
    REPEAT GEN_TAC
    THEN PURE_REWRITE_TAC[NextState;R_COND;RUNNING;S_COND;
       WATCHOUT;OVERTEMP;WFAILED;DEMAND;IDLE;ILLEGAL;
       Xor2_DEF;True_DEF;False_DEF;Not_DEF]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[]
    THEN CONV_TAC (DEPTH_CONV let_CONV)
    THEN REWRITE_TAC[] THEN COND_CASES_TAC
    THEN REWRITE_TAC[Zero_Eq_0;Arb_True;PWM]);;

% Shutdown transition condition 4 %
let SCOND4_lemma = prove_thm(`SCOND4_lemma`,
  "!(t:time) s m v p.
      (s=True,Arb,Arb,Arb,Arb,Arb,True,
         Arb,Arb,Arb,ArbNot0,Arb,Arb)\/
      (s=True,Arb,Arb,Arb,Arb,Arb,False,
         Arb,Arb,Arb,Zero,Arb,Arb) ==>
      (m=Cruise)=>(NextState(s,m,v,p)t=(Shutdown,T,Close))|
                  (NextState(s,m,v,p)t=(Shutdown,v,Close))",
    REPEAT GEN_TAC
    THEN PURE_REWRITE_TAC[NextState;R_COND;RUNNING;S_COND;
       WATCHOUT;OVERTEMP;WFAILED;DEMAND;IDLE;ILLEGAL;
       Xor2_DEF;True_DEF;False_DEF;Not_DEF]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[]
    THEN CONV_TAC (DEPTH_CONV let_CONV)
    THEN REWRITE_TAC[] THEN COND_CASES_TAC
    THEN REWRITE_TAC[PWM;Eq_DEF;Zero] THEN BETA_TAC
    THEN REWRITE_TAC[ArbNot0_Not_Zero]);;

% Idle transition condition 1 %
let ICOND1_lemma = prove_thm(`ICOND1_lemma`,
  "!(t:time) s m v p.
      ((m=Cruise) /\ (s=True,False,Arb,False,False,False,
                          True,Arb,Arb,Arb,Zero,Arb,Arb))
     ==> (NextState(s,m,v,p)t=
             (Idle,T,PWM(p,ANGLE s t,^idledem)))",
    PURE_REWRITE_TAC[NextState;R_COND;S_COND;I_COND;RUNNING;
       WATCHOUT;OVERTEMP;WFAILED;DEMAND;IDLE;ILLEGAL;BRAKE;
```

```
        GEAR;CRUISE;Xor2_DEF;True_DEF;False_DEF]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[]
    THEN CONV_TAC (DEPTH_CONV let_CONV)
    THEN REWRITE_TAC[Not_DEF] THEN BETA_TAC
    THEN REWRITE_TAC[Zero_Eq_0]
    THEN REWRITE_TAC[Mode_dist2]);;

% Idle transition condition 2 %
let ICOND2_lemma = prove_thm('ICOND2_lemma',
  "!(t:time) s m v p.
      ((m=Cruise) /\ (s=True,Arb,True,False,False,False,
                         True,Arb,Arb,Arb,Zero,Arb,Arb))
   ==> (NextState(s,m,v,p)t=
       (Idle,T,PWM(p,ANGLE s t,~idledem)))",
    PURE_REWRITE_TAC[NextState;R_COND;S_COND;I_COND;RUNNING;
       WATCHOUT;OVERTEMP;WFAILED;DEMAND;IDLE;ILLEGAL;BRAKE;
       GEAR;CRUISE;Xor2_DEF;True_DEF;False_DEF]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[]
    THEN CONV_TAC (DEPTH_CONV let_CONV)
    THEN REWRITE_TAC[Not_DEF] THEN BETA_TAC
    THEN REWRITE_TAC[Zero_Eq_0;Mode_dist2]);;

% Idle transition condition 3 %
let ICOND3_lemma = prove_thm('ICOND3_lemma',
  "!(t:time) s m v p.
     (m=Manual) /\
       ((s=True,False,False,False,False,False,
           True,Arb,Arb,Arb,Zero,Arb,Arb)\/
        (s=True,False,True,False,False,False,
           True,Arb,Arb,Arb,Zero,Arb,Arb) \/
        (s=True,True,True,False,False,False,
           True,Arb,Arb,Arb,Zero,Arb,Arb))
    ==> (NextState(s,m,v,p)t=
      (Idle,v,PWM(p,ANGLE s t,^idledem))",
    PURE_REWRITE_TAC[NextState;R_COND;S_COND;I_COND;RUNNING]
    THEN PURE_REWRITE_TAC[WATCHOUT;OVERTEMP;WFAILED;DEMAND]
    THEN PURE_REWRITE_TAC[IDLE;ILLEGAL;BRAKE;GEAR;CRUISE]
    THEN PURE_REWRITE_TAC[Xor2_DEF;True_DEF;False_DEF]
    THEN BETA_TAC THEN REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[]
    THEN CONV_TAC (DEPTH_CONV let_CONV)
    THEN REWRITE_TAC[Not_DEF] THEN BETA_TAC
    THEN REWRITE_TAC[Zero_Eq_0;Mode_Distinct;Mode_dist3]);;

% Cruise transition condition 1 %
let CCOND1_lemma = prove_thm(`CCOND1_lemma`,
  "!(t:time) s m v p.
    ((m=Manual) /\
     (s=True,True,False,False,False,False,False,
        Arb,True,Arb,ArbNot0,ArbNot0,Arb))
   ==>(NextState(s,m,v,p)t=
        (Cruise,v,PWM(p,ANGLE s t,SPEED s t)))",
     REPEAT GEN_TAC
     THEN PURE_REWRITE_TAC[NextState;R_COND;S_COND;I_COND;
        C_COND;RUNNING;WATCHOUT;OVERTEMP;WFAILED;DEMAND;
        SPEED;RESUME;IDLE;ILLEGAL;BRAKE;GEAR;CRUISE;
        Xor2_DEF;True_DEF;False_DEF;Not_DEF]
     THEN BETA_TAC THEN REPEAT STRIP_TAC
     THEN ASM_REWRITE_TAC[]
     THEN CONV_TAC (DEPTH_CONV let_CONV)
     THEN REWRITE_TAC[Eq_DEF] THEN BETA_TAC
     THEN REWRITE_TAC[Mode_Distinct;Mode_dist3;
         ArbNot0_Not_Zero]);;

% Cruise transition condition 2 %
let CCOND2_lemma = prove_thm(`CCOND2_lemma`,
  "!(t:time) s m v p.
     ((m=Idle) /\
      (s=True,True,False,False,False,False,
         True,Arb,Arb,True,Zero,Arb,Arb))
   ==>(NextState(s,m,T,p)t=
        (Cruise,T,PWM(p,ANGLE s t,SPEED s t)))",
     PURE_REWRITE_TAC[NextState;R_COND;S_COND;
       I_COND;C_COND;RUNNING;WATCHOUT;OVERTEMP;WFAILED;
       DEMAND;SPEED;RESUME;IDLE;ILLEGAL;BRAKE;GEAR;CRUISE;
       Xor2_DEF;True_DEF;False_DEF;Not_DEF]
     THEN BETA_TAC THEN REPEAT STRIP_TAC
     THEN ASM_REWRITE_TAC[]
     THEN CONV_TAC (DEPTH_CONV let_CONV)
     THEN REWRITE_TAC[Zero_Eq_0;Mode_Distinct]);;


% Traction transition condition %
let TCOND_lemma = prove_thm(`TCOND_lemma`,
  "!(t:time) s m v p.
       ((m=Manual) /\
        (s=True,False,Arb,False,False,False,False,
           True,Arb,Arb,ArbNot0,Zero,Arb)) ==>
   (NextState(s,m,v,p)t=
      (Traction,v,PWM(p,ANGLE s t,DEMAND s t)))",
     REPEAT GEN_TAC
     THEN PURE_REWRITE_TAC[NextState;R_COND;S_COND;I_COND;
       C_COND;T_COND;WATCHOUT;OVERTEMP;WFAILED;DEMAND;SPEED;
       RESUME;IDLE;ILLEGAL;BRAKE;GEAR;CRUISE;TRACTION;
       RUNNING;Xor2_DEF;True_DEF;False_DEF;Not_DEF]
     THEN BETA_TAC THEN REPEAT STRIP_TAC
     THEN ASM_REWRITE_TAC[]
     THEN CONV_TAC (DEPTH_CONV let_CONV)
     THEN REWRITE_TAC[Mode_Distinct;ArbNot0_Not_Zero]
     THEN REWRITE_TAC[Mode_dist3;Eq_DEF]
     THEN BETA_TAC THEN REWRITE_TAC[ArbNot0_Not_Zero;Zero]
```

```
    THEN BETA_TAC);;


% Manual transition condition %
let MCOND_lemma = prove_thm(`MCOND_lemma`,
  "!(t:time) s m v p.
    (~(m=Shutdown) /\
      (s=True,Arb,Arb,False,False,False,False,False,
         False,False,ArbNot0,Arb,Arb)) ==>
    (NextState(s,m,v,p)t=
       (Manual,F,PWM(p,ANGLE s t,(DEMAND s)t)))",
    REPEAT STRIP_TAC
    THEN PURE_REWRITE_TAC[NextState;R_COND;S_COND;I_COND;
       C_COND;T_COND;WATCHOUT;OVERTEMP;WFAILED;DEMAND;SPEED;
       RESUME;IDLE;ILLEGAL;BRAKE;GEAR;CRUISE;TRACTION;
       RUNNING;Xor2_DEF;True_DEF;False_DEF;Not_DEF]
    THEN BETA_TAC THEN ASM_REWRITE_TAC[]
    THEN CONV_TAC(DEPTH_CONV let_CONV)
    THEN REWRITE_TAC[True_DEF;False_DEF;Eq_DEF;
         ArbNot0_Not_Zero]
    THEN BETA_TAC THEN BOOL_CASES_TAC "Arb t:bool"
    THEN REWRITE_TAC[]);;


%------------------------------------------------------------
%    Deterministic properties of the Finite State Machine
%------------------------------------------------------------


new_type_abbrev (`OpSig`,":Mode#bool#Dir");;


let Init = new_definition (`Init_DEF`,
  "Init (ip,op) = !t. INIT(ip,op)t");;


let Safe = new_definition (`Safe_DEF`,
  "Safe (ip,op) = !t. SAFE(ip,op)t");;
```

```
let Nextstate = new_definition ('Nextstate_DEF',
  "Nextstate (s,m,v,p) ((s':^sig_ty),m',v',p') =
     !t. NextState(s,m,v,p)t = (m',v',p')");;


let FSM = prove_thm ('FSM',
  "(!ip op. Init(ip,op) ==> Safe(ip,op)) /\
   (!ip op ip' op'.
     Nextstate (ip,op)(ip',op') /\ Safe(ip,op) ==>
         Safe(ip',op')) ==>(!(e:time->^sig_ty).
         LSA(Init,Nextstate)e=PLSA(Init,Safe,Nextstate)e)",
      STRIP_TAC
      THEN POP_ASSUM (MP_TAC o GEN_ALL o (SPECL[
         "(ip:time->^sig_ty)t";"(op:time->OpSig)t";
         "(ip:time->^sig_ty)(SUC t)";"(op:time->OpSig)
          (SUC t)"]))
      THEN POP_ASSUM (MP_TAC o GEN_ALL o (SPECL[
         "(ip:time->^sig_ty)t";"(op:time->OpSig)t"]))
      THEN REWRITE_TAC [LSA;PLSA] THEN REPEAT STRIP_TAC
      THEN EQ_TAC THENL [STRIP_TAC
              THEN EXISTS_TAC "s:time->OpSig"
              THEN ASM_REWRITE_TAC[] THEN INDUCT_TAC
              THENL [RES_TAC;
                     RES_TAC THEN POP_ASSUM(\th.ALL_TAC)
                     THEN POP_ASSUM MP_TAC
                     THEN ASM_REWRITE_TAC[]];
              STRIP_TAC THEN EXISTS_TAC "s:time->OpSig"
              THEN ASM_REWRITE_TAC[]]);;
```

Annex D: Formal Specification of Solid Modelling

D.1 Definitions

✓ COMP_DEF $\vdash \forall s.\ \mathsf{COMP}\ s = \mathsf{UNIV\ DIFF}\ s$

✓ METRIC_DEF $\vdash \forall r.\ \mathsf{METRIC}\ r = (\forall x\ y. 0 \leq r\ x\ y) \wedge (\forall x\ y. r\ x\ y = r\ y\ x) \wedge$
$(\forall x\ y. (x = y) \supset (r\ x\ y = 0)) \wedge (\forall x\ y\ z. r\ x\ z \leq r\ x\ y + r\ y\ z)$

DISTANCE_DEF $\vdash \forall x\ y.\ \mathsf{DISTANCE}\ x\ y = ((x = y) \Rightarrow 0 \mid 1)$

d

✗ OPEN_BALL_DEF $\vdash \forall s\ x\ R.\ \mathsf{OPEN\_BALL}\ s\ x\ R =$
$\{y \mid \exists d. y\ \mathsf{IN}\ s \wedge \mathsf{METRIC}\ d \supset d\ x\ y \leq R\}$

✗ ✓ OPEN_DEF $\vdash \forall s.\ \mathsf{OPEN}\ s = (\forall x. (\exists R. x\ \mathsf{IN\ OPEN\_BALL}\ s\ x\ R))$

is.

✗ NEIGHBOUR_DEF $\vdash \forall x\ s.\ \mathsf{NEIGHBOUR}\ (x, s) = \{y \mid y\ \mathsf{IN}\ s \wedge (\exists y'. y' = x)\}$

✗ ✓ CLOSED_DEF $\vdash \forall s.\ \mathsf{CLOSED}\ s = \mathsf{OPEN}\ (\mathsf{COMP}\ s)$

✗ LIMIT_DEF $\vdash \forall x\ s.\ \mathsf{LIMIT}\ (x, s) = (\forall s'. (s' = \mathsf{NEIGHBOUR}\ (x, s')) \wedge (\exists y. y\ \mathsf{IN}\ s' \wedge$
$\neg(y = x)))$

limit always false

✗ ✓ CLOSURE_DEF $\vdash \forall s.\ \mathsf{CLOSURE}\ s = s\ \mathsf{UNION}\ \{x \mid \mathsf{LIMIT}\ (x, s)\}$ CLOSURE S = S

✗ ✓ INTERIOR_DEF $\vdash \forall s.\ \mathsf{INTERIOR}\ s = \{x \mid \exists t. \mathsf{OPEN}\ t \wedge t\ \mathsf{SUBSET}\ s \wedge x\ \mathsf{IN}\ t\}$

✗ ✓ INSIDE_DEF $\vdash \forall x\ s.\ \mathsf{INSIDE}\ (x, s) = x\ \mathsf{IN\ INTERIOR}\ s$

✗ ✓ BOUNDARY_DEF $\vdash \forall s.\ \mathsf{BOUNDARY}\ s =$
$\{x \mid x\ \mathsf{IN\ CLOSURE}\ s \wedge x\ \mathsf{IN\ CLOSURE}\ (\mathsf{COMP}\ s)\}$ = {}

S x in S

✗ ✓ ON_DEF $\vdash \forall x\ s.\ \mathsf{ON}\ (x, s) = x\ \mathsf{IN\ BOUNDARY}\ s$ ON(x,S) always false

✗ ✓ THIN_DEF $\vdash \forall s.\ \mathsf{THIN}\ s = (\mathsf{INTERIOR}\ (\mathsf{CLOSURE}\ s) = \{\ \})$

THINBOUND_DEF $\vdash \forall s.\, \mathsf{THINBOUND}\ s = \mathsf{THIN}\ (\mathsf{BOUNDARY}\ s)$

REGULAR_DEF $\vdash \forall s.\, \mathsf{REGULAR}\ s = \mathsf{CLOSURE}\ (\mathsf{INTERIOR}\ s)$

REGULARISED_DEF $\vdash \forall s.\, \mathsf{REGULARISED}\ s = (s = \mathsf{REGULAR}\ s)$

REG_UNION_DEF $\vdash \forall s\ t.\, s\ \mathsf{REG\_UNION}\ t = \mathsf{REGULAR}\ (s\ \mathsf{UNION}\ t)$

REG_INTER_DEF $\vdash \forall s\ t.\, s\ \mathsf{REG\_INTER}\ t = \mathsf{REGULAR}\ (s\ \mathsf{INTER}\ t)$

REG_DIFF_DEF $\vdash \forall s\ t.\, s\ \mathsf{REG\_DIFF}\ t = \mathsf{REGULAR}\ (s\ \mathsf{DIFF}\ t)$

REG_COMP_DEF $\vdash \forall s.\, \mathsf{REG\_COMP}\ s = \mathsf{REGULAR}\ (\mathsf{COMP}\ s)$

in_DEF $\vdash \forall t\ s.\, t\ \mathsf{in}\ s = t\ \mathsf{REG\_INTER}\ \mathsf{INTERIOR}\ s$

on_DEF $\vdash \forall t\ s.\, t\ \mathsf{on}\ s = t\ \mathsf{REG\_INTER}\ \mathsf{BOUNDARY}\ s$

out_DEF $\vdash \forall t\ s.\, t\ \mathsf{out}\ s = t\ \mathsf{REG\_INTER}\ \mathsf{COMP}\ s$

point_TY_DEF $\vdash \exists rep.\, \mathsf{TYPE\_DEFINITION}\ (\mathsf{TRP}\ (\lambda v\ tl.\, (\exists n_0\ n_1\ n_2.\, v = n_0, n_1, n_2) \wedge$
$(\mathsf{LENGTH}\ tl = 0)))\ rep$

point_ISO_DEF $\vdash (\forall a.\, \mathsf{ABS\_point}\ (\mathsf{REP\_point}\ a) = a) \wedge$
$(\forall r.\, \mathsf{TRP}\ (\lambda v\ tl.\, (\exists n_0\ n_1\ n_2.\, v = n_0, n_1, n_2) \wedge$
$(\mathsf{LENGTH}\ tl = 0))\ r = (\mathsf{REP\_point}\ (\mathsf{ABS\_point}\ r) = r))$

POINT_DEF $\vdash \forall n_0\ n_1\ n_2.\, \mathsf{POINT}\ n_0\ n_1\ n_2 = \mathsf{ABS\_point}\ (\mathsf{Node}\ (n_0, n_1, n_2)\ [\,])$

X_DEF $\vdash \forall x\ y\ z.\, \mathsf{X}\ (\mathsf{POINT}\ x\ y\ z) = x$

Y_DEF $\vdash \forall x\ y\ z.\, \mathsf{Y}\ (\mathsf{POINT}\ x\ y\ z) = y$

Z_DEF $\vdash \forall x\ y\ z.\, \mathsf{Z}\ (\mathsf{POINT}\ x\ y\ z) = z$

object_TY_DEF $\vdash \exists rep.\, \mathsf{TYPE\_DEFINITION}\ (\mathsf{TRP}\ (\lambda v\ tl.\, (\exists s_0\ s_1.\, v = s_0, s_1) \wedge$
$(\mathsf{LENGTH}\ tl = 0)))\ rep$

object_ISO_DEF $\vdash (\forall a.\, \mathsf{ABS\_object}\ (\mathsf{REP\_object}\ a) = a) \wedge$
$(\forall r.\, \mathsf{TRP}\ (\lambda v\ tl.\, (\exists s_0\ s_1.\, v = s_0, s_1) \wedge$
$(\mathsf{LENGTH}\ tl = 0))\ r = (\mathsf{REP\_object}\ (\mathsf{ABS\_object}\ r) = r))$

OBject_DEF $\vdash \forall s_0\ s_1.\, \mathsf{OBject}\ s_0\ s_1 = \mathsf{ABS\_object}\ (\mathsf{Node}\ (s_0, s_1)\ [\,])$

b_DEF $\vdash \forall interior\ bound.\, \mathsf{b}\ (\mathsf{OBject}\ interior\ bound) = bound$

i_DEF $\vdash \forall interior\ bound.\, \mathsf{i}\ (\mathsf{OBject}\ interior\ bound) = interior$

Closure_DEF $\vdash \forall A.\, \mathsf{Closure}\ A = \mathsf{b}\ A\ \mathsf{UNION}\ \mathsf{i}\ A$

HAS_CARD_DEF $\vdash \forall A\ n.\ \mathsf{HAS\_CARD}\ A\ n = (\mathsf{CARD}\ A = n)$

OBJECT_DEF $\vdash \forall A.\ \mathsf{OBJECT}\ A = \mathsf{CLOSED}\ (\mathsf{Closure}\ A)\ \wedge\ \neg(\mathsf{b}\ A = \{\ \})\ \wedge$
$(\exists n.\ \mathsf{HAS\_CARD}\ (\mathsf{Closure}\ A)\ n)$

OBJ_DEF $\vdash \forall A.\ \mathsf{OBJ}\ A = (\mathsf{OBject}\ (\mathsf{i}\ A)\ (\mathsf{b}\ A))$

DOT_DEF $\vdash \forall D.\ \mathsf{DOT}\ D = \mathsf{OBJECT}\ D\ \wedge\ (\mathsf{CARD}\ (\mathsf{b}\ D) = 1)\ \wedge\ (\mathsf{i}\ D = \{\ \})$

CURVE_DEF $\vdash \forall C.\ \mathsf{CURVE}\ C = \mathsf{OBJECT}\ C\ \wedge\ \neg(\mathsf{i}\ C = \{\ \})\ \wedge\ (\mathsf{FINITE}\ (\mathsf{b}\ C))$

OPEN_CURVE_DEF $\vdash \forall C.\ \mathsf{OPEN\_CURVE}\ C = \mathsf{CURVE}\ C\ \wedge\ (\mathsf{HAS\_CARD}\ (\mathsf{b}\ C)\ 1)$

SURFACE_DEF $\vdash \forall s.\ \mathsf{SURFACE}\ s = \mathsf{OBJECT}\ s\ \wedge\ \neg(\mathsf{i}\ s = \{\ \})\ \wedge$
$(\exists\, C.\ \mathsf{CLOSED\_CURVE}\ C\ \wedge\ (\mathsf{Closure}\ C\ \mathsf{SUBSET}\ (\mathsf{b}\ s)))$

PRIMITIVE_DEF $\vdash \forall p.\ \mathsf{PRIMITIVE}\ p = \mathsf{OBJECT}\ p\ \wedge\ (\forall s_1\ s_2.\ \mathsf{SURFACE}\ s_1\ \wedge$
$\mathsf{SURFACE}\ s_2 \supset (\mathsf{DISJOINT}\ (\mathsf{Closure}\ s_1)\ (\mathsf{Closure}\ s_2)))$

HOLLOWED_DEF $\vdash \forall k.\ \mathsf{HOLLOWED}\ k = \mathsf{PRIMITIVE}\ k\ \wedge\ (\forall s_1\ s_2.\ \mathsf{SURFACE}\ s_1\ \wedge$
$\mathsf{SURFACE}\ s_2 \supset (\mathsf{DISJOINT}\ (\mathsf{Closure}\ s_1)\ (\mathsf{Closure}\ s_2)))$

PLANE_DEF $\vdash \forall p.\ \mathsf{PLANE}\ p = \mathsf{SURFACE}\ p\ \wedge\ (\forall h.\ (h\ \mathsf{IN}\ (\mathsf{Closure}\ p)) \supset (\exists k\ \ l\ m\ n.\ k *$
$\mathsf{X}\ (h) + l * \mathsf{Y}\ (h) + m * \mathsf{Z}\ (h) = n))$

FAR_SPACE_DEF $\vdash \forall f\ k\ l\ m\ n.\ \mathsf{FAR\_SPACE}\ f(k,l,m,n) = \mathsf{SURFACE}\ f\ \wedge$
$(\forall x.\ (x\ \mathsf{IN}\ \mathsf{Closure}\ f)) \supset (k * \mathsf{X}\ (x) + l * \mathsf{Y}\ (x) + m * \mathsf{Z}\ (x) \geq n)$

NEAR_SPACE_DEF $\vdash \forall f\ k\ l\ m\ n.\ \mathsf{NEAR\_SPACE}\ f(k,l,m,n) = \mathsf{SURFACE}\ f\ \wedge$
$(\forall x.\ (x\ \mathsf{IN}\ \mathsf{Closure}\ f)) \supset (k * \mathsf{X}\ (x) + l * \mathsf{Y}\ (x) + m * \mathsf{Z}\ (x) \leq n)$

BLOCK_DEF $\vdash \forall dx\ dy\ dz;\ l\ w\ h.\ \mathsf{BLOCK}\ (dx,dy,dz)\ (l,w,h) =$
$\{p \mid \exists e.\ \mathsf{PRIMITIVE}\ e\ \wedge\ (p\ \mathsf{IN}\ e)\ \wedge$
$(\mathsf{FAR\_SPACE}\ (\mathsf{b}\ e)\ (1,0,0,dx)\ \wedge\ (\mathsf{NEAR\_SPACE}\ (\mathsf{b}\ e)\ (1,0,0,l+dx)\ \wedge$
$(\mathsf{FAR\_SPACE}\ (\mathsf{b}\ e)\ (0,1,0,dy)\ \wedge\ (\mathsf{NEAR\_SPACE}\ (\mathsf{b}\ e)\ (0,1,0,w+dy)\ \wedge$
$(\mathsf{FAR\_SPACE}\ (\mathsf{b}\ e)\ (0,0,1,dz)\ \wedge\ (\mathsf{NEAR\_SPACE}\ (\mathsf{b}\ e)\ (0,0,1,h+dz)))))))\}$

SQR_DEF $\vdash \forall x.\ \mathsf{SQR}\ x = x * x$

CYLINDER_X_DEF $\vdash \forall dx\ dy\ dz\ r\ h.\ \mathsf{CYLINDER\_X}\ (dx,dy,dz)\ (r,h) =$
$\{p \mid \exists c.\ \mathsf{PRIMITIVE}\ c\ \wedge\ (p\ \mathsf{IN}\ c)\ \wedge$
$(\mathsf{NEAR\_SPACE}\ (\mathsf{b}\ c)\ (1,0,0,dx)\ \wedge\ (\mathsf{FAR\_SPACE}\ (\mathsf{b}\ c)\ (1,0,0,h+dx)\ \wedge$
$(\mathsf{SQR}\ (\mathsf{Y}\ (p) + dy) + \mathsf{SQR}\ (\mathsf{Z}\ (p) + dz) \leq \mathsf{SQR}\ (r)))))\}$

CYLINDER_Y_DEF $\vdash \forall dx\ dy\ dz\ r\ h.\ \mathsf{CYLINDER\_Y}\ (dx, dy, dz)\ (r, h) =$

$\{p \mid \exists c.\ \mathsf{PRIMITIVE}\ c\ \wedge\ (p\ \mathsf{IN}\ c)\ \wedge$

$(\mathsf{NEAR\_SPACE}\ (\mathsf{b}\ c)\ (0, 1, 0, dy)\ \wedge\ (\mathsf{FAR\_SPACE}\ (\mathsf{b}\ c)\ (0, 1, 0, h + dy)\ \wedge$

$(\mathsf{SQR}\ (\mathsf{X}\ (p) + dx) + \mathsf{SQR}\ (\mathsf{Z}\ (p) + dz) \leq \mathsf{SQR}\ (r)))))\}$

CYLINDER_Z_DEF $\vdash \forall dx\ dy\ dz\ r\ h.\ \mathsf{CYLINDER\_Z}\ (dx, dy, dz)\ (r, h) =$

$\{p \mid \exists c.\ \mathsf{PRIMITIVE}\ c\ \wedge\ (p\ \mathsf{IN}\ c)\ \wedge$

$(\mathsf{NEAR\_SPACE}\ (\mathsf{b}\ c)\ (0, 0, 1, dz)\ \wedge\ (\mathsf{FAR\_SPACE}\ (\mathsf{b}\ c)\ (0, 0, 1, h + dz)\ \wedge$

$(\mathsf{SQR}\ (\mathsf{X}\ (p) + dx) + \mathsf{SQR}\ (\mathsf{Y}\ (p) + dy) \leq \mathsf{SQR}\ (r)))))\}$

BUILD_DEF $\vdash (\forall f.\ \mathsf{BUILD}\ [] = \lambda f.\ \{\ \})\ \wedge\ (\forall f.\ \mathsf{BUILD}\ (\mathsf{CONS}\ hd\ tl) = \lambda f.\ hd\ f\ \mathsf{BUILD}\ [tl])$

NULL_DETECT_DEF $\vdash \forall s.\ = ((\mathsf{BUILD}\ s\ \mathsf{REG\_INTER}) = \{\ \})$

within_DEF $\vdash \forall n\ k\ p.\ \mathsf{within}\ n\ (k, p) = ((k \leq p)\ \wedge\ (k \leq n)\ \wedge\ (n \leq p))$

OVERALL_B_DEF $\vdash \forall dx\ dy\ dz\ l\ w\ h.\ \mathsf{OVERALL\_B}\ (dx, dy, dz)\ (l, w, h) =$

$\{p \mid (\mathsf{X}\ (p)\ \mathsf{within}\ (dx, l + dx))\ \wedge\ (\mathsf{Y}\ (p)\ \mathsf{within}\ (dy, w + dy))\ \wedge$

$(\mathsf{Z}\ (p)\ \mathsf{within}\ (dz, h + dz))\}$

OVERALL_CX_DEF $\vdash \forall dx\ dy\ dz\ r\ h.$

$\mathsf{OVERALL\_CX}\ (dx, dy, dz)\ (r, h) = \mathsf{OVERALL\_B}\ (dx, dy, dz)\ (h, 2 * r, 2 * r)$

OVERALL_CY_DEF $\vdash \forall dx\ dy\ dz\ r\ h.$

$\mathsf{OVERALL\_CY}\ (dx, dy, dz)\ (r, h) = \mathsf{OVERALL\_B}\ (dx, dy, dz)\ (2 * r, h, 2 * r)$

OVERALL_CZ_DEF $\vdash \forall dx\ dy\ dz\ r\ h.$

$\mathsf{OVERALL\_CZ}\ (dx, dy, dz)\ (r, h) = \mathsf{OVERALL\_B}\ (dx, dy, dz)\ (2 * r, 2 * r, h)$

D.2 Theorems

DIFF_UNION $\vdash \forall a\ c\ d.\ a\ \mathsf{DIFF}\ (c\ \mathsf{UNION}\ d) = a\ \mathsf{DIFF}\ (d\ \mathsf{UNION}\ c)$

AND_DISTR $\vdash \forall x\ y\ z.\ (x\ \vee\ y)\ \wedge\ z = x\ \wedge\ z\ \vee\ y\ \wedge\ z$

UNION_SUBSET $\vdash \forall a\ c\ d.\ (a\ \mathsf{UNION}\ c)\ \mathsf{SUBSET}\ d \supset a\ \mathsf{SUBSET}\ d\ \wedge\ c\ \mathsf{SUBSET}\ d$

SUBSET_ABSORP $\vdash \forall a\ c.\ a\ \mathsf{SUBSET}\ c \supset (a\ \mathsf{UNION}\ c = c)$

COMP_UNIV $\vdash \forall s.\ s\ \mathsf{UNION}\ \mathsf{COMP}\ s = \mathsf{UNIV}$

COMP_DISJOINT $\vdash \forall s.\ s\ \mathsf{INTER}\ \mathsf{COMP}\ s = \{\ \}$

COMP_UNION $\vdash \forall s\ t.\ \mathsf{COMP}\ (s\ \mathsf{UNION}\ t) = \mathsf{COMP}\ s\ \mathsf{INTER}\ \mathsf{COMP}\ t$

`COMP_INTER` $\vdash \forall s\ t.\ \mathrm{COMP}\ (s\ \mathrm{INTER}\ t) = \mathrm{COMP}\ s\ \mathrm{UNION}\ \mathrm{COMP}\ t$

`COMP_DIFF` $\vdash \forall s\ t.\ \mathrm{COMP}\ (s\ \mathrm{DIFF}\ t) = \mathrm{COMP}\ s\ \mathrm{UNION}\ t$

`COMP_EQ` $\vdash \forall s\ t.\ (s = t) \supset (\mathrm{COMP}\ s = \mathrm{COMP}\ t)$

`COMP_IDEM` $\vdash \forall s.\ \mathrm{COMP}\ (\mathrm{COMP}\ s) = s$

`METRIC_DISTANCE` $\vdash \mathrm{METRIC\ DISTANCE}$

`OPEN_EMPTY` $\vdash \mathrm{OPEN}\ \{\,\}$

`DIST_REFL` $\vdash \forall x\ d.\ \mathrm{METRIC}\ d \supset (d\ x\ x = 0)$

`METRIC_lemma` $\vdash \forall s\ x'.\ (\exists R\ d.\ x'\ \mathrm{IN}\ s\ \wedge\ \mathrm{METRIC}\ d \supset d\ x'\ x' \leq R)$

`OPEN_UNIV` $\vdash \mathrm{OPEN\ UNIV}$

`OPEN_INTER` $\vdash \forall s\ t.\ \mathrm{OPEN}\ s\ \wedge\ \mathrm{OPEN}\ t \supset \mathrm{OPEN}\ (s\ \mathrm{INTER}\ t)$

`OPEN_UNION` $\vdash \forall s\ t.\ \mathrm{OPEN}\ s\ \wedge\ \mathrm{OPEN}\ t \supset \mathrm{OPEN}\ (s\ \mathrm{UNION}\ t)$

`CLOSED_EMPTY` $\vdash \mathrm{CLOSED}\ \{\,\}$

`CLOSED_UNIV` $\vdash \mathrm{CLOSED\ UNIV}$

`CLOSED_INTER` $\vdash \forall s\ t.\ \mathrm{CLOSED}\ s\ \wedge\ \mathrm{CLOSED}\ t \supset \mathrm{CLOSED}\ (s\ \mathrm{INTER}\ t)$

`CLOSED_UNION` $\vdash \forall s\ t.\ \mathrm{CLOSED}\ s\ \wedge\ \mathrm{CLOSED}\ t \supset \mathrm{CLOSED}\ (s\ \mathrm{UNION}\ t)$

`IN_LIMIT` $\vdash \forall s.\ \mathrm{CLOSED}\ s = (\forall x.\ \mathrm{LIMIT}\ (x, s) \supset x\ \mathrm{IN}\ s)$

`IN_CLOSURE` $\vdash \forall x\ s.\ x\ \mathrm{IN\ CLOSURE}\ s \supset \neg(s\ \mathrm{INTER\ NEIGHBOUR}\ (x, s) = \{\,\})$

`CLOSURE_CLOSED` $\vdash \forall s.\ \mathrm{CLOSED}\ s = (s = \mathrm{CLOSURE}\ s)$

`LIMIT_REFL` $\vdash \forall s.\ s\ \mathrm{UNION}\ \{x \mid \mathrm{LIMIT}\ (x, s)\} = s$

`CLOSURE_SUBSET` $\vdash \forall s\ t.\ s\ \mathrm{SUBSET}\ t \supset \mathrm{CLOSURE}\ s\ \mathrm{SUBSET\ CLOSURE}\ t$

`CLOSURE_UNION` $\vdash \forall s\ t.\ \mathrm{CLOSURE}\ (s\ \mathrm{UNION}\ t) = \mathrm{CLOSURE}\ s\ \mathrm{UNION\ CLOSURE}\ t$

`CLOSURE_lemma` $\vdash \forall s\ t.\ \mathrm{CLOSURE}\ (s\ \mathrm{INTER}\ t)\ \mathrm{SUBSET}\ (\mathrm{CLOSURE}\ s\ \mathrm{INTER\ CLOSURE}\ t)$

`BOUNDED` $\vdash \forall x\ s.\ \mathrm{INSIDE}\ (x, s) \supset (s = \mathrm{NEIGHBOUR}\ (x, s))$

`INTERIOR_DISJOINT` $\vdash \forall s\ x.\ \neg(x\ \mathrm{IN\ INTERIOR}\ s\ \wedge\ x\ \mathrm{IN\ COMP}\ (\mathrm{INTERIOR}\ s))$

`INTERIOR_IDEM` $\vdash \forall s.\ \mathrm{INTERIOR}\ s = s$

`INTERIOR_EMPTY` $\vdash \forall s.\ (\mathrm{INTERIOR}\ s = \{\,\}) \supset (s = \{\,\})$

INTERIOR_OPEN $\vdash \forall s.\,(s = \mathsf{INTERIOR}\ s) = \mathsf{OPEN}\ s$

INTERIOR_SUBSET $\vdash \forall s\ t.\ s\ \mathsf{SUBSET}\ t \supset \mathsf{INTERIOR}\ s\ \mathsf{SUBSET}\ \mathsf{INTERIOR}\ t$

EXISTS_OPEN $\vdash \forall s\ t.\,(\exists t'.\,\mathsf{OPEN}\ t' \ \wedge\ (\forall x'.\,x'\ \mathsf{IN}\ t' \supset x'\ \mathsf{IN}\ s\ \vee\ x'\ \mathsf{IN}\ t))$

INTERIOR_INTER $\vdash \forall s\ t.\,\mathsf{INTERIOR}\,(s\ \mathsf{INTER}\ t) = \mathsf{INTERIOR}\ s\ \mathsf{INTER}\ \mathsf{INTERIOR}\ t$

INTERIOR_UNION $\vdash \forall s\ t.\,(\mathsf{INTERIOR}\ s\ \mathsf{UNION}\ \mathsf{INTERIOR}\ t)\ \mathsf{SUBSET}$
$\mathsf{INTERIOR}\,(s\ \mathsf{UNION}\ t)$

INTERIOR_CLOSURE $\vdash \forall s.\,\mathsf{INTERIOR}\ s\ \mathsf{SUBSET}\ \mathsf{INTERIOR}\,(\mathsf{CLOSURE}\ s)$

INTERIOR_DIFF $\vdash \forall s\ t.\,\mathsf{INTERIOR}\,(s\ \mathsf{DIFF}\ t) = \mathsf{INTERIOR}\ s\ \mathsf{DIFF}\ \mathsf{INTERIOR}\ t$

BOUNDARY_lemma $\vdash \forall s.\,\mathsf{BOUNDARY}\ s = \mathsf{CLOSURE}\ s\ \mathsf{INTER}\ \mathsf{CLOSURE}\,(\mathsf{COMP}\ s)$

BOUNDARY_CLOSED $\vdash \forall s.\,\mathsf{CLOSED}\,(\mathsf{BOUNDARY}\ s)$

BOUNDARY_COMP $\vdash \forall s.\,\mathsf{BOUNDARY}\ s = \mathsf{BOUNDARY}\,(\mathsf{COMP}\ s)$

CLOSURE_BOUNDED $\vdash \forall s.\,\mathsf{CLOSURE}\ s = s\ \mathsf{UNION}\ \mathsf{BOUNDARY}\ s$

CLOSURE_IDEM $\vdash \forall s.\,\mathsf{CLOSURE}\ s = s$

CLOSURE_INTER $\vdash \forall s\ t.\,\mathsf{CLOSURE}\,(s\ \mathsf{INTER}\ t) = \mathsf{CLOSURE}\ s\ \mathsf{INTER}\ \mathsf{CLOSURE}\ t$

CLOSURE_lemma1 $\vdash \forall s.\,\mathsf{CLOSURE}\ s = \mathsf{INTERIOR}\ s\ \mathsf{UNION}\ \mathsf{BOUNDARY}\ s$

INTERIOR_COMP $\vdash \forall s.\,\mathsf{INTERIOR}\ s = \mathsf{COMP}\,(\mathsf{CLOSURE}\,(\mathsf{COMP}\ s))$

COMP_INTERIOR $\vdash \forall s.\,\mathsf{COMP}\,(\mathsf{INTERIOR}\ s) = \mathsf{CLOSURE}\,(\mathsf{COMP}\ s)$

COMP_CLOSURE $\vdash \forall s.\,\mathsf{COMP}\,(\mathsf{CLOSURE}\ s) = \mathsf{INTERIOR}\,(\mathsf{COMP}\ s)$

CLOSURE_DISJOINT $\vdash \forall s.\,\mathsf{DISJOINT}\,(\mathsf{INTERIOR}\ s)\,(\mathsf{BOUNDARY}\ s)$

UNIV_COMPOSIT $\vdash \forall s\ t.\,\mathsf{INTERIOR}\ s\ \mathsf{UNION}\,(\mathsf{BOUNDARY}\ s\ \mathsf{UNION}$
$\mathsf{INTERIOR}\,(\mathsf{COMP}\ s)) = \mathsf{UNIV}$

BOUNDARY_UNION_SUBSET $\vdash \forall s\ t.\,\mathsf{BOUNDARY}\,(s\ \mathsf{UNION}\ t)\ \mathsf{SUBSET}$
$(\mathsf{BOUNDARY}\ s\ \mathsf{UNION}\ \mathsf{BOUNDARY}\ t)$

BOUNDARY_INTER_SUBSET $\vdash \forall s\ t.\,\mathsf{BOUNDARY}\,(s\ \mathsf{INTER}\ t)\ \mathsf{SUBSET}$
$(\mathsf{BOUNDARY}\ s\ \mathsf{UNION}\ \mathsf{BOUNDARY}\ t)$

BOUNDARY_DIFF_SUBSET $\vdash \forall s\ t.\,\mathsf{BOUNDARY}\,(s\ \mathsf{DIFF}\ t)\ \mathsf{SUBSET}$
$(\mathsf{BOUNDARY}\ s\ \mathsf{UNION}\ \mathsf{BOUNDARY}\ t)$

INTERIOR_UNION_SUBSET $\vdash \forall s\ t.$ INTERIOR (s UNION t) SUBSET
(INTERIOR s UNION (INTERIOR t UNION
(BOUNDARY s INTER BOUNDARY t)))

BOUNDARY_UNION $\vdash \forall s\ t.$ BOUNDARY (s UNION t) =
(BOUNDARY s INTER INTERIOR (COMP t)) UNION
((INTERIOR (COMP s) INTER BOUNDARY t) UNION
(BOUNDARY s INTER (BOUNDARY t INTER
CLOSURE (COMP s INTER COMP t))))

BOUNDARY_INTER $\vdash \forall s\ t.$ BOUNDARY (s INTER t) =
(BOUNDARY s INTER INTERIOR t) UNION
((INTERIOR s INTER BOUNDARY t) UNION
(BOUNDARY s INTER (BOUNDARY t INTER
CLOSURE (s INTER t))))

BOUNDARY_DIFF $\vdash \forall s\ t.$ BOUNDARY (s DIFF t) =
(BOUNDARY s INTER INTERIOR (COMP t)) UNION
((INTERIOR s INTER BOUNDARY t) UNION
(BOUNDARY s INTER (BOUNDARY t INTER
CLOSURE (s INTER COMP t))))

CLOSED_BOUNDARY_INTER1 $\vdash \forall s\ t.$ CLOSED s $\wedge$ CLOSED t $\supset$
(BOUNDARY (s INTER t) =
(BOUNDARY s INTER INTERIOR t) UNION
((INTERIOR s INTER BOUNDARY t) UNION
(BOUNDARY s INTER BOUNDARY t)))

CLOSED_BOUNDARY_INTER2 $\vdash \forall s\ t.$ CLOSED s $\wedge$ CLOSED t $\supset$
(BOUNDARY (s INTER t) = (s INTER BOUNDARY t) UNION
(BOUNDARY s INTER t))

THIN_SUBSET $\vdash \forall s\ t.$ THIN s $\wedge$ t SUBSET s $\supset$ THIN t

THINBOUND_CLOSED $\vdash \forall s.$ CLOSED s $\supset$ THINBOUND s

THINBOUND_OPEN $\vdash \forall s.$ OPEN s $\supset$ THINBOUND s

THIN_UNION $\vdash \forall s\ t.$ THIN s $\wedge$ THIN t $\supset$ THIN (s UNION t)

THIN_INTER $\vdash \forall s\ t.$ THIN s $\wedge$ THIN t $\supset$ THIN (s INTER t)

THIN_DIFF $\vdash \forall s\, t.$ THIN s $\wedge$ THIN $t \supset$ THIN (s DIFF t)

THINBOUND_UNION_SUBSET1 $\vdash \forall s\, t.$ THINBOUND s $\wedge$ THINBOUND $t \supset$
INTERIOR (s UNION t) SUBSET CLOSURE (INTERIOR s UNION INTERIOR t)

THINBOUND_UNION_SUBSET2 $\vdash \forall s\, t.$ THINBOUND s $\wedge$ THINBOUND $t \supset$
(CLOSURE (INTERIOR (s UNION t)) =
CLOSURE (INTERIOR s UNION INTERIOR t))

THINBOUND_INTER_SUBSET1 $\vdash \forall s\, t.$ THINBOUND s $\wedge$ THINBOUND $t \supset$
INTERIOR (CLOSURE s INTER CLOSURE t) SUBSET CLOSURE (s INTER t)

THINBOUND_INTER_SUBSET2 $\vdash \forall s\, t.$ THINBOUND s $\wedge$ THINBOUND $t \supset$
(INTERIOR (CLOSURE (s INTER t)) =
INTERIOR (CLOSURE s INTER CLOSURE t))

REGULAR_CLOSED $\vdash \forall s.$ CLOSED (REGULAR s)

THIN_REGULAR $\vdash \forall s.$ THINBOUND (REGULAR s)

REGULAR_EMPTY $\vdash \forall s.$ REGULARISED s $\wedge$ (INTERIOR $s = \{\,\}) \supset (s = \{\,\})$

REGULAR_IDEM $\vdash \forall s.$ REGULAR $s = s$

REGULAR_REFL $\vdash \forall s.$ REGULARISED (REGULAR s)

IMP_LEFT $\vdash \forall a\, b\, c.\, (a \supset b \wedge c) \supset (a \supset b)$

IN_IMP1 $\vdash \forall s\, t\, t'.\, (\forall x.\, x$ IN $t' \supset x$ IN s $\wedge$ x IN $t) \supset (\forall x.\, x$ IN $t' \supset x$ IN $s)$

IN_IMP2 $\vdash \forall s\, t\, t'.\, (\forall x.\, x$ IN $t' \supset x$ IN s $\wedge$ x IN $t) \supset (\forall x.\, x$ IN $t' \supset x$ IN $t)$

REGULAR_INTER $\vdash \forall s\, t.$ REGULAR (s INTER t) = REGULAR s INTER REGULAR t

INTERIOR_REG_INTER $\vdash \forall s\, t.$ INTERIOR (s REG_INTER t) =
INTERIOR (REGULAR s INTER REGULAR t)

REGULAR_REG_INTER $\vdash \forall s\, t.\, s$ REG_INTER t = REGULAR s REG_INTER REGULAR t

REGULAR_UNION $\vdash \forall s\, t.$ REGULAR (s UNION t) = REGULAR s UNION REGULAR t

REGULARISED_REG_INTER $\vdash \forall s\, t.$ REGULARISED s $\wedge$ REGULARISED $t \supset$
(INTERIOR (s REG_INTER t) = INTERIOR s INTER INTERIOR t)

REGULARISED_REG_UNION $\vdash \forall s\, t.$ REGULARISED s $\wedge$
REGULARISED $t \supset$ (s REG_UNION t = s UNION t)

REGULARISED_REG_DIFF $\vdash \forall s\, t.$ REGULARISED s $\wedge$
REGULARISED $t \supset$ (s REG_DIFF t = s REG_INTER REG_COMP t)

REG_COMP_lemma $\vdash \forall s.$ CLOSED $s \supset$ (REG_COMP s = CLOSURE (COMP s))

REG_DIFF_lemma $\vdash \forall s\, t.$ s REG_DIFF t = s REG_INTER COMP t

COMP_REG_UNION $\vdash \forall s\, t.$ COMP (s REG_UNION t) = COMP s INTER COMP t

INTERIOR_REG_COMP $\vdash \forall s\, t.$ INTERIOR (REG_COMP s) = COMP s

BOUNDARY_REG_COMP $\vdash \forall s\, t.$ BOUNDARY (REG_COMP s) = BOUNDARY s

COMP_REG_COMP $\vdash \forall s\, t.$ COMP (REG_COMP s) = INTERIOR s

INTERIOR_REG_DIFF $\vdash \forall s\, t.$ INTERIOR (s REG_DIFF t) = INTERIOR s INTER COMP t

BOUNDARY_REG_UNION $\vdash \forall s\, t.$ BOUNDARY (s REG_UNION t) =
(BOUNDARY s INTER COMP t) UNION
((COMP t INTER BOUNDARY t) UNION
(BOUNDARY s INTER (BOUNDARY t INTER
CLOSURE (COMP s INTER COMP t))))

BOUNDARY_REG_INTER $\vdash \forall s\, t.$ BOUNDARY (s REG_INTER t) =
(BOUNDARY s INTER INTERIOR t) UNION
((BOUNDARY t INTER INTERIOR s) UNION
(BOUNDARY s INTER (BOUNDARY t INTER
CLOSURE (INTERIOR s INTER CLOSURE t))))

BOUNDARY_REG_DIFF $\vdash \forall s\, t.$ BOUNDARY (s REG_DIFF t) =
(CLOSURE s INTER BOUNDARY (COMP t)) UNION
((INTERIOR s INTER BOUNDARY t) UNION
(BOUNDARY s INTER (BOUNDARY t INTER
CLOSURE (INTERIOR s INTER COMP t))))

MEMBER_lemma $\vdash \forall s\, t.$ t = (t in s) UNION ((t on s) UNION (t out s))

MEMBER_DISJOINT $\vdash \forall s\, t.$ ((t in s) REG_INTER (t on s) = { }) $\wedge$
((t on s) REG_INTER (t out s) = { }) $\wedge$ ((t in s) REG_INTER (t out s) = { })

IN_SUBSET $\vdash \forall s\, t.$ (t in s) SUBSET REGULAR (INTERIOR s)

ON_SUBSET $\vdash \forall s\, t.$ (t on s) SUBSET REGULAR (BOUNDARY s)

OUT_SUBSET $\vdash \forall s\, t.$ (t out s) SUBSET REGULAR (COMP s)

MEMBER_INTER $\vdash \forall s\ s_1\ s_2\ t.(s = s_1$ REG_INTER $s_2) \supset$
$(t$ in $s = (t$ in $s_1)$ REG_INTER $(t$ in $s_2))$

Point_Axiom $\vdash \forall f.(\exists\forall fn.(\forall n_0\ n_1\ n_2.fn$ (POINT $n_0\ n_1\ n_2) = f\ n_0\ n_1\ n_2))$

Object_Axiom $\vdash \forall f.(\exists\forall fn.(\forall s_0\ s_1.fn$ (OBject $s_0\ s_1) = f\ s_0\ s_1))$

EXISTS_OBJECT $\vdash \exists A.$ OBJECT A

FINITE_OBJECT $\vdash \forall A.$ OBJECT $A \supset (\neg($Closure $A = \{\ \}))$

UNIQUE_OBJECT $\vdash \forall\ M\ N.$ OBJECT $M\ \wedge$ OBJECT $N \supset (((\mathrm{b}\ (M) = \mathrm{b}\ (N))\ \wedge$
$(\mathrm{i}\ (M) = \mathrm{i}\ (N))) \supset ($OBJ $M =$ OBJ$N))$

DISJOINT_MEMBER $\vdash \forall A\ B.$
(DISJOINT (Closure A) (Closure B)) $\supset ((\mathrm{i}\ A$ INTER i $B = \{\ \})\ \wedge$
$(\mathrm{b}\ A$ INTER i $B = \{\ \})\ \wedge\ (\mathrm{i}\ A$ INTER b $B = \{\ \})\ \wedge\ (\mathrm{b}\ A$ INTER b $B = \{\ \}))$

NON_OVERLAP_ELEM $\vdash \forall A.$ OBJECT $A \supset (\mathrm{i}\ A\ CONSTINTER\ \mathrm{b}\ A = \{\ \})$

DOT_NOT_CURVE $\vdash \forall D.$ DOT $D \supset \neg($CURVE $D)$

DOT_NOT_SURFACE $\vdash \forall D.$ DOT $D \supset \neg($SURFACE $D)$

NON_EMPTY $\vdash \forall p.$ PRIMITIVE $p \supset (\exists a.\ a$ IN (Closure p))

ENCLOSED $\vdash \forall p\ r.(p$ IN (Closure r)) $= ((p$ IN (b r)) $\vee$ (p IN (i r)))

BOUNDED $\vdash \forall P.$ PRIMITIVE $P \supset \neg($Closure $P = \{\ \})$

CONNECTED $\vdash \forall P \exists C.$ PRIMITIVE $P\ \wedge$ CURVE $C\ \wedge$
((Closure C) SUBSET (Closure P)) $\supset (\forall A\ B.((A$ IN (Closure C)) $\wedge$
(B IN (Closure C))) $\supset ((A$ IN (Closure P)) $\wedge$ (B IN (Closure P))))

DOT_NOT_PRIMITIVE $\vdash \forall D.$ DOT $D \supset \neg($PRIMITIVE $D)$

POINT_INCLUSION $\vdash \forall a\ p.\neg(a$ IN (Closure p)) $\vee$ (a IN (b p)) $\vee$ (a IN (i p))

well_order1 $\vdash \forall n\ a\ b\ c\ d.((n$ within $(a,b))\ \wedge\ (n$ within $(c,d))) \supset (((a \leq c)\ \wedge$
$(d \leq b)) \Rightarrow (n$ within $(a,b))\ |\ (n$ within $(c,d)))$

well_order2 $\vdash \forall n\ a\ b\ c\ d.((n$ within $(a,b))\ \vee\ (n$ within $(c,d))) \supset (((a \leq c)\ \wedge$
$(d \leq b)) \Rightarrow (n$ within $(a,b))\ |\ (n$ within $(c,d)))$

well_order3 $\vdash \forall n\ a\ b\ c\ d.((n$ within $(a,b))\ \wedge\ \neg(n$ within $(c,d))) \supset ((c \leq b)\ \wedge$
$(d \leq b)) \Rightarrow ((n$ within $(a,c))\ \wedge\ (n$ within $(d,b)))\ |\ ((c \leq b)\ \wedge$
$(b \leq d)) \Rightarrow (n$ within $(a,c))\ |\ (n$ within $(a,b))$

D.3 Procedures of Proofs and Tactics

```
%<-------------------------------------------------------------
   General theorems and tactics
 -------------------------------------------------------------->%

let DIFF_UNION = prove_thm (`DIFF_UNION`,
  "!(a:(*)set) c d. a DIFF (c UNION d) = a DIFF (d UNION c)",
    REPEAT GEN_TAC
    THEN ONCE_DEPTH_FST_REWRITE_TAC[UNION_COMM]
    THEN REFL_TAC);;


let AND_DISTR = prove_thm (`AND_DISTR`,
  "!x y z. (x \/ y) /\ z = (x /\ z) \/ (y /\ z)",
    REPEAT GEN_TAC THEN BOOL_CASES_TAC "x:bool"
    THENL[REWRITE_TAC[] THEN BOOL_CASES_TAC "y:bool"
          THEN REWRITE_TAC[];REWRITE_TAC[]]);;


let UNION_SUBSET = prove_thm (`UNION_SUBSET`,
  "!(a:(*)set) c d. (a UNION c) SUBSET d
          ==> (a SUBSET d) /\ (c SUBSET d)",
    REWRITE_TAC[SUBSET_DEF;IN_UNION]
    THEN REPEAT STRIP_TAC THEN RES_TAC);;


let SUBSET_ABSORP = prove_thm (`SUBSET_ABSORP`,
  "!(a:(*)set) c. a SUBSET c ==> (a UNION c = c)",
     REWRITE_TAC[EXTENSION;SUBSET_DEF;IN_UNION]
     THEN REPEAT STRIP_TAC THEN EQ_TAC
     THEN STRIP_TAC THENL[RES_TAC;ASM_REWRITE_TAC[]]);;


%<-------------------------------------------------------------
   Extension definitions and theorems for set algebra
 -------------------------------------------------------------->%

let COMP = new_definition (`COMP_DEF`,
  "COMP (s:(*)set) = UNIV DIFF s");;
```

```
let COMP_UNIV = prove_thm  ('COMP_UNIV',
  "!(s:(*)set). s UNION (COMP s) = UNIV",
     GEN_TAC
     THEN REWRITE_TAC[COMP;EXTENSION;IN_UNION;IN_DIFF]
     THEN STRIP_TAC THEN EQ_TAC
     THENL [STRIP_TAC THEN REWRITE_TAC[IN_UNIV];
            REWRITE_TAC[IN_UNIV;EXCLUDED_MIDDLE]]);;

let COMP_DISJOINT = prove_thm ('COMP_DISJOINT',
  "!(s:(*)set). s INTER (COMP s) = {}",
     GEN_TAC
     THEN REWRITE_TAC[COMP;EXTENSION;IN_INTER;IN_DIFF]
     THEN STRIP_TAC THEN EQ_TAC
     THENL [REWRITE_TAC[IN_UNIV]
              THEN STRIP_TAC THEN RES_TAC;
              STRIP_TAC THEN REWRITE_TAC[IN_UNIV]
              THEN BOOL_CASES_TAC "(x IN s):bool"
              THEN ASSUME_TAC NOT_IN_EMPTY THEN RES_TAC]);;

let COMP_UNION = prove_thm ('COMP_UNION',
  "!(s:(*)set) (t:(*)set).
     COMP(s UNION t) = COMP(s) INTER COMP(t)",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[COMP;EXTENSION;IN_DIFF;IN_UNION;
        IN_INTER;IN_DIFF;IN_UNIV]
    THEN GEN_TAC THEN REWRITE_TAC[DE_MORGAN_THM]);;

let COMP_INTER = prove_thm ('COMP_INTER',
  "!(s:(*)set) (t:(*)set).
     COMP(s INTER t) = COMP(s) UNION COMP(t)",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[COMP;EXTENSION;IN_DIFF;IN_UNION]
    THEN REWRITE_TAC[IN_INTER;IN_DIFF;IN_UNIV]
    THEN REWRITE_TAC[DE_MORGAN_THM]);;


let COMP_DIFF = prove_thm (`COMP_DIFF`,
  "!(s:(*)set) (t:(*)set).
     COMP(s DIFF t) = COMP(s) UNION t",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[COMP;EXTENSION;IN_DIFF;IN_UNION]
    THEN REWRITE_TAC[IN_DIFF;IN_UNIV;DE_MORGAN_THM]);;


let COMP_EQ = prove_thm (`COMP_EQ`,
  "!(s:(*)set) (t:(*)set). (s=t) ==> ((COMP s)=(COMP t))",
    REPEAT STRIP_TAC THEN ONCE_ASM_REWRITE_TAC[]
    THEN REFL_TAC);;


let COMP_IDEM = prove_thm (`COMP_IDEM`,
  "!(s:(*)set). COMP(COMP s)= s",
    REWRITE_TAC[COMP;EXTENSION;IN_DIFF;IN_UNIV]);;


%<---------------------------------------------------------
   Topological space definitions and theorems
 --------------------------------------------------------->%

let METRIC = new_definition (`METRIC_DEF`,
  "METRIC (r:*->*->num) =
     (!x y. 0<=(r x y)) /\
     (!x y. (r x y)=(r y x)) /\
     (!x y. (x=y) ==> ((r x y) =0)) /\
     (!x y z. (r x z) <= (r x y) + (r y z))");;


%<---------------------------------------------------------
   A METRIC example : 2-D grid in first quadrant
 --------------------------------------------------------->%

let DISTANCE = new_definition (`DISTANCE_DEF`,
  "DISTANCE x y = (x=y) => 0 | 1");;
```

```
let METRIC_DISTANCE = prove_thm (`METRIC_DISTANCE`,
  "METRIC(DISTANCE:*->*->num)",
    REWRITE_TAC[DISTANCE;METRIC]
    THEN REPEAT STRIP_TAC
    THENL [COND_CASES_TAC
      THENL [REWRITE_TAC[LESS_EQ_REFL];
        REWRITE_TAC[ZERO_LESS_EQ]];COND_CASES_TAC
        THENL [ONCE_ASM_REWRITE_TAC[] THEN REWRITE_TAC[];
        COND_CASES_TAC
        THENL [POP_ASSUM (\th1. POP_ASSUM(\th2.
          ASSUME_TAC th1 THEN ASSUME_TAC th2))
          THEN POP_ASSUM MP_TAC THEN ONCE_ASM_REWRITE_TAC[]
          THEN REWRITE_TAC[IMP_DISJ_THM];REFL_TAC]];
           ONCE_ASM_REWRITE_TAC[]
           THEN REWRITE_TAC[IMP_CLAUSES];
           COND_CASES_TAC
           THENL [REWRITE_TAC[ZERO_LESS_EQ];
             COND_CASES_TAC
             THENL [COND_CASES_TAC
               THENL [ASSUM_LIST(\t.ASSUME_TAC(
                 ONCE_REWRITE_RULE[(el 1 t)] (el 2 t)))
                 THEN RES_TAC;
                   REWRITE_TAC[ADD_CLAUSES;LESS_EQ_REFL]];
                   COND_CASES_TAC
      THENL[REWRITE_TAC[ADD_CLAUSES;LESS_EQ_REFL];
        REWRITE_TAC[LESS_EQ_ADD]]]]]]);;


let OPEN_BALL = new_definition (`OPEN_BALL_DEF`,
  "OPEN_BALL s x R =
     {y | ?d. ((y IN s) /\ METRIC(d))==> (d x y)<=R}");;


let OPEN = new_definition (`OPEN_DEF`,
  "OPEN s = !x. ?R. x IN OPEN_BALL s x R");;
```

```
let OPEN_EMPTY = prove_thm (`OPEN_EMPTY`,
  "OPEN (EMPTY:(*)set)",
     REWRITE_TAC[OPEN;OPEN_BALL;NOT_IN_EMPTY]
     THEN GEN_TAC THEN CONV_TAC SET_SPEC_CONV);;

let DIST_REFL = prove_thm (`DIST_REFL`,
  "!x d. METRIC(d) ==> ((d x x)=0)",
     REPEAT GEN_TAC THEN REWRITE_TAC[METRIC]
     THEN REPEAT STRIP_TAC THEN POP_ASSUM (\th. ALL_TAC)
     THEN POP_ASSUM MP_TAC
     THEN CONV_TAC LEFT_IMP_FORALL_CONV
     THEN EXISTS_TAC "x"
     THEN CONV_TAC LEFT_IMP_FORALL_CONV
     THEN EXISTS_TAC "x" THEN ASSUME_TAC EQ_REFL
     THEN ONCE_ASM_REWRITE_TAC[] THEN REWRITE_TAC[]);;

let METRIC_lemma = prove_thm (`METRIC_lemma`,
  "! (s:(*)set) x'. ?R d. x' IN s /\ METRIC d
        ==> (d x' x') <= R",
    REPEAT GEN_TAC THEN EXISTS_TAC "R"
    THEN EXISTS_TAC "d" THEN REPEAT STRIP_TAC
    THEN ASSUME_TAC DIST_REFL THEN RES_TAC
    THEN ASM_REWRITE_TAC[ZERO_LESS_EQ]);;

let OPEN_UNIV = prove_thm (`OPEN_UNIV`,
  "OPEN (UNIV:(*)set)",
     REWRITE_TAC[OPEN;OPEN_BALL;IN_UNIV]
     THEN GEN_TAC THEN EXISTS_TAC "R"
     THEN CONV_TAC SET_SPEC_CONV THEN EXISTS_TAC "d"
     THEN STRIP_TAC THEN ASSUME_TAC DIST_REFL
     THEN RES_TAC THEN ASM_REWRITE_TAC[ZERO_LESS_EQ]);;

let OPEN_INTER = prove_thm (`OPEN_INTER`,
  "!(s:(*)set) (t:(*)set).
```

```
      (OPEN s) /\ (OPEN t) ==> (OPEN (s INTER t))",
    REPEAT GEN_TAC THEN REWRITE_TAC[OPEN;OPEN_BALL]
    THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
    THEN REWRITE_TAC[IN_INTER] THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "R" THEN EXISTS_TAC "d"
    THEN REPEAT STRIP_TAC THEN ASSUME_TAC DIST_REFL
    THEN RES_TAC THEN ASM_REWRITE_TAC[ZERO_LESS_EQ]);;

let OPEN_UNION = prove_thm (`OPEN_UNION`,
  "!(s:(*)set) (t:(*)set).
      (OPEN s) /\ (OPEN t) ==> (OPEN (s UNION t))",
    REPEAT GEN_TAC THEN REWRITE_TAC[OPEN;OPEN_BALL]
    THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
    THEN REWRITE_TAC[IN_UNION] THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "R" THEN EXISTS_TAC "d"
    THEN REPEAT STRIP_TAC THEN ASSUME_TAC DIST_REFL
    THEN RES_TAC THEN ASM_REWRITE_TAC[ZERO_LESS_EQ]);;

%<-----------------------------------------------------------
   Closed-set definitions and theorems
 ------------------------------------------------------------>%

let NEIGHBOUR = new_definition (`NEIGHBOUR_DEF`,
  "NEIGHBOUR (x,(s:(*)set)) = {y | (y IN s) /\ (?y. y=x)}");;

let CLOSED = new_definition (`CLOSED_DEF`,
  "CLOSED (s:(*)set) = OPEN (COMP s)");;

let CLOSED_EMPTY = prove_thm (`CLOSED_EMPTY`,
  "CLOSED (EMPTY:(*)set)",
   REWRITE_TAC[CLOSED;OPEN;COMP;OPEN_BALL;IN_DIFF]
   THEN GEN_TAC THEN EXISTS_TAC "R"
   THEN CONV_TAC SET_SPEC_CONV THEN EXISTS_TAC "d"
   THEN REWRITE_TAC[IN_UNIV;NOT_IN_EMPTY]
   THEN STRIP_TAC THEN ASSUME_TAC DIST_REFL
```

```
    THEN RES_TAC THEN ASM_REWRITE_TAC[ZERO_LESS_EQ]);;

let CLOSED_UNIV = prove_thm (`CLOSED_UNIV`,
  "CLOSED (UNIV:(*)set)",
    REWRITE_TAC[CLOSED;OPEN;COMP;OPEN_BALL;IN_DIFF]
    THEN GEN_TAC THEN EXISTS_TAC "R"
    THEN CONV_TAC SET_SPEC_CONV THEN REWRITE_TAC[IN_UNIV]);;

let CLOSED_INTER = prove_thm (`CLOSED_INTER`,
  "!(s:(*)set) (t:(*)set).
      (CLOSED s) /\ (CLOSED t) ==> CLOSED (s INTER t)",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[CLOSED;OPEN;OPEN_BALL;
           COMP;IN_DIFF;IN_UNIV]
    THEN REPEAT GEN_TAC
    THEN CONV_TAC (DEPTH_CONV SET_SPEC_CONV)
    THEN REWRITE_TAC[IN_INTER] THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "R" THEN EXISTS_TAC "d"
    THEN REPEAT STRIP_TAC THEN ASSUME_TAC DIST_REFL
    THEN RES_TAC THEN ASM_REWRITE_TAC[ZERO_LESS_EQ]);;

let CLOSED_UNION = prove_thm (`CLOSED_UNION`,
  "!(s:(*)set) (t:(*)set).
      (CLOSED s) /\ (CLOSED t) ==> CLOSED (s UNION t)",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[CLOSED;OPEN;OPEN_BALL;
           COMP;IN_DIFF;IN_UNIV]
    THEN REPEAT GEN_TAC
    THEN CONV_TAC (DEPTH_CONV SET_SPEC_CONV)
    THEN REWRITE_TAC[IN_UNION] THEN REPEAT STRIP_TAC
    THEN EXISTS_TAC "R" THEN EXISTS_TAC "d"
    THEN REPEAT STRIP_TAC THEN ASSUME_TAC DIST_REFL
    THEN RES_TAC THEN ASM_REWRITE_TAC[ZERO_LESS_EQ]);;
```

```
let LIMIT = new_definition ('LIMIT_DEF',
  "LIMIT (x,(s:(*)set)) =
    !(s:(*)set). (s=NEIGHBOUR(x,s)) /\
        (?y. y IN s /\ (y=x))");;


let IN_LIMIT = prove_thm ('IN_LIMIT',
  "!(s:(*)set). CLOSED(s) = (!x. LIMIT(x,s) ==> (x IN s))",
     GEN_TAC
     THEN REWRITE_TAC[CLOSED;LIMIT;OPEN;COMP;NEIGHBOUR]
     THEN REWRITE_TAC[OPEN_BALL;IN_DIFF;IN_UNIV;EXTENSION]
     THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
     THEN EQ_TAC
     THENL [CONV_TAC(DEPTH_CONV LEFT_IMP_FORALL_CONV)
       THEN EXISTS_TAC "x"
       THEN CONV_TAC(DEPTH_CONV LEFT_IMP_EXISTS_CONV)
       THEN CONV_TAC(DEPTH_CONV LEFT_IMP_EXISTS_CONV)
       THEN REPEAT STRIP_TAC THEN EXISTS_TAC "s"
       THEN REPEAT STRIP_TAC
       THEN ASSUM_LIST(\th. ASSUME_TAC(
          REWRITE_RULE[(el 1 th)](el 2 th)))
       THEN POP_ASSUM MATCH_ACCEPT_TAC;
       CONV_TAC(DEPTH_CONV LEFT_IMP_FORALL_CONV)
       THEN EXISTS_TAC "x"
       THEN CONV_TAC(DEPTH_CONV LEFT_IMP_EXISTS_CONV)
       THEN REPEAT STRIP_TAC THEN EXISTS_TAC "R"
       THEN EXISTS_TAC "d" THEN REPEAT STRIP_TAC
       THEN ASSUME_TAC DIST_REFL
       THEN RES_TAC THEN ASM_REWRITE_TAC[ZERO_LESS_EQ]]);;


%<-------------------------------------------------------
  Closure definitions and theorems
 ------------------------------------------------------->%

let CLOSURE = new_definition ('CLOSURE_DEF',
  "CLOSURE (s:(*)set) = s UNION {x| LIMIT(x,s)}");;
```

```
let IN_CLOSURE = prove_thm ('IN_CLOSURE',
  "!x (s:(*)set). (x IN CLOSURE s) ==>
          ~(s INTER NEIGHBOUR(x,s)={})",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[CLOSURE;NEIGHBOUR;EXTENSION;IN_UNION]
    THEN REWRITE_TAC[IN_INTER;NOT_IN_EMPTY]
    THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
    THEN REWRITE_TAC[DE_MORGAN_THM]
    THEN STRIP_TAC
    THENL [CONV_TAC NOT_FORALL_CONV
      THEN EXISTS_TAC "x" THEN ASM_REWRITE_TAC[DE_MORGAN_THM]
      THEN EXISTS_TAC "x" THEN REFL_TAC;
      POP_ASSUM MP_TAC THEN REWRITE_TAC[LIMIT]
      THEN CONV_TAC LEFT_IMP_FORALL_CONV
      THEN EXISTS_TAC "s" THEN STRIP_TAC
      THEN CONV_TAC NOT_FORALL_CONV
      THEN EXISTS_TAC "x" THEN REWRITE_TAC[DE_MORGAN_THM]
      THEN ASSUM_LIST(
          \th.ASSUME_TAC(REWRITE_RULE[(el 1 th)](el 2 th)))
      THEN ASM_REWRITE_TAC[] THEN EXISTS_TAC "x"
      THEN REFL_TAC]);;

let CLOSURE_CLOSED = prove_thm ('CLOSURE_CLOSED',
  "!(s:(*)set). (CLOSED s) = (s=CLOSURE s)",
    GEN_TAC
    THEN REWRITE_TAC[EXTENSION;CLOSURE;CLOSED;OPEN;
         COMP;OPEN_BALL;IN_DIFF;IN_UNION;IN_UNIV;LIMIT]
    THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV) THEN EQ_TAC
    THENL [REPEAT STRIP_TAC THEN EQ_TAC
      THENL [REPEAT STRIP_TAC THEN ASM_REWRITE_TAC[];
        REWRITE_TAC[NEIGHBOUR;EXTENSION]
        THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
   THEN REPEAT STRIP_TAC THEN POP_ASSUM MP_TAC
   THEN CONV_TAC LEFT_IMP_FORALL_CONV
      THEN EXISTS_TAC "s" THEN REPEAT STRIP_TAC
   THEN ASSUM_LIST(\th.ASSUME_TAC(
          REWRITE_RULE[(el 1 th)](el 2 th)))
   THEN ASM_REWRITE_TAC[]];
        REWRITE_TAC[NEIGHBOUR]
        THEN CONV_TAC(DEPTH_CONV LEFT_IMP_FORALL_CONV)
   THEN EXISTS_TAC "x" THEN REPEAT STRIP_TAC
   THEN EXISTS_TAC "R" THEN EXISTS_TAC "d"
   THEN ASSUME_TAC DIST_REFL THEN REPEAT STRIP_TAC
   THEN RES_TAC THEN ASM_REWRITE_TAC[ZERO_LESS_EQ]]);;

let LIMIT_REFL = prove_thm (`LIMIT_REFL`,
  "!(s:(*)set). s UNION {x| LIMIT(x,s)} = s",
    GEN_TAC
    THEN REWRITE_TAC[LIMIT;EXTENSION;IN_UNION;NEIGHBOUR]
    THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
    THEN GEN_TAC THEN EQ_TAC
    THENL [REPEAT STRIP_TAC THEN POP_ASSUM MP_TAC
           THEN CONV_TAC LEFT_IMP_FORALL_CONV
           THEN EXISTS_TAC "s" THEN REPEAT STRIP_TAC
           THEN ASSUM_LIST(
           \th.ASSUME_TAC(REWRITE_RULE[(el 1 th)](el 2 th)))
           THEN POP_ASSUM MATCH_ACCEPT_TAC;
           REPEAT STRIP_TAC THEN ASM_REWRITE_TAC[]]);;

let CLOSURE_SUBSET = prove_thm (`CLOSURE_SUBSET`,
  "!(s:(*)set) (t:(*)set).
      (s SUBSET t) ==> ((CLOSURE s) SUBSET (CLOSURE t))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[CLOSURE;SUBSET_DEF;IN_UNION;LIMIT]
    THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
    THEN REPEAT STRIP_TAC
    THENL [RES_TAC THEN ASM_REWRITE_TAC[];
```

```
      POP_ASSUM MP_TAC THEN CONV_TAC LEFT_IMP_FORALL_CONV
      THEN EXISTS_TAC "s" THEN REPEAT STRIP_TAC
      THEN ASSUM_LIST(
        \th.ASSUME_TAC(REWRITE_RULE[(el 1 th)](el 2 th)))
      THEN RES_TAC THEN ASM_REWRITE_TAC[]]);;

let CLOSURE_UNION = prove_thm (`CLOSURE_UNION`,
  "!(s:(*)set) (t:(*)set).
      CLOSURE (s UNION t) = ((CLOSURE s) UNION (CLOSURE t))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[CLOSURE;EXTENSION;IN_UNION;LIMIT]
    THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
    THEN GEN_TAC THEN EQ_TAC
    THENL [REPEAT STRIP_TAC
      THENL [ASM_REWRITE_TAC[]; ASM_REWRITE_TAC[];
 POP_ASSUM MP_TAC
      THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
 THEN CONV_TAC(DEPTH_CONV LEFT_IMP_FORALL_CONV)
      THEN REWRITE_TAC[IN_UNION]
      THEN EXISTS_TAC "s" THEN REPEAT STRIP_TAC
 THEN ASSUM_LIST(\th.ASSUME_TAC(
         REWRITE_RULE[(el 1 th)](el 2 th)))
 THEN ASM_REWRITE_TAC[]];REWRITE_TAC[NEIGHBOUR]
      THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
 THEN REWRITE_TAC[IN_UNION] THEN REPEAT STRIP_TAC
 THENL [ASM_REWRITE_TAC[];
      POP_ASSUM MP_TAC THEN CONV_TAC LEFT_IMP_FORALL_CONV
      THEN EXISTS_TAC "s" THEN REPEAT STRIP_TAC
      THEN ASSUM_LIST(\th.ASSUME_TAC(
        REWRITE_RULE[(el 1 th)](el 2 th)))
 THEN ASM_REWRITE_TAC[];
 ASM_REWRITE_TAC[];POP_ASSUM MP_TAC
      THEN CONV_TAC LEFT_IMP_FORALL_CONV
      THEN EXISTS_TAC "s" THEN REPEAT STRIP_TAC
 THEN ASSUM_LIST(\th.ASSUME_TAC(
        REWRITE_RULE[(el 1 th)](el 2 th)))
 THEN ASM_REWRITE_TAC[]]]);;


let CLOSURE_lemma = prove_thm (`CLOSURE_lemma`,
  "!(s:(*)set) (t:(*)set).
      CLOSURE (s INTER t) SUBSET
           ((CLOSURE s) INTER (CLOSURE t))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[CLOSURE;SUBSET_DEF;IN_UNION]
    THEN REWRITE_TAC[IN_INTER;IN_UNION;LIMIT]
    THEN CONV_TAC (DEPTH_CONV SET_SPEC_CONV)
    THEN REPEAT STRIP_TAC
    THENL [ASM_REWRITE_TAC[];ASM_REWRITE_TAC[];
   POP_ASSUM MP_TAC THEN REWRITE_TAC[NEIGHBOUR;EXTENSION]
   THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
   THEN REWRITE_TAC[IN_INTER]
   THEN CONV_TAC (DEPTH_CONV LEFT_IMP_FORALL_CONV)
          THEN EXISTS_TAC "s" THEN REPEAT STRIP_TAC
   THEN ASSUM_LIST(\th.ASSUME_TAC(
               REWRITE_RULE[(el 1 th)](el 2 th)))
   THEN ASM_REWRITE_TAC[];
   POP_ASSUM MP_TAC THEN REWRITE_TAC[NEIGHBOUR;EXTENSION]
   THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
   THEN REWRITE_TAC[IN_INTER]
   THEN CONV_TAC (DEPTH_CONV LEFT_IMP_FORALL_CONV)
          THEN EXISTS_TAC "t" THEN REPEAT STRIP_TAC
   THEN ASSUM_LIST(\th.ASSUME_TAC(
               REWRITE_RULE[(el 1 th)](el 2 th)))
   THEN ASM_REWRITE_TAC[]]);;


%<-----------------------------------------------------
   Interior definitions and theorems
 -----------------------------------------------------<%
```

```
let INTERIOR =  new_definition (`INTERIOR_DEF`,
  "INTERIOR (s:(*)set) =
     {x| ?(t:(*)set). OPEN t /\ (t SUBSET s) /\ (x IN t)}");;


let INSIDE = new_definition (`INSIDE_DEF`,
  "INSIDE (x,(s:(*)set)) = x IN INTERIOR s");;


let BOUNDED = prove_thm (`BOUNDED`,
  "!x (s:(*)set). INSIDE(x,s) ==> (s=NEIGHBOUR(x,s))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[INSIDE;NEIGHBOUR;INTERIOR;EXTENSION]
    THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
    THEN CONV_TAC LEFT_IMP_EXISTS_CONV
    THEN REWRITE_TAC[SUBSET_DEF]
    THEN REPEAT STRIP_TAC THEN EQ_TAC
    THENL [REPEAT STRIP_TAC THEN ASM_REWRITE_TAC[]
   THEN EXISTS_TAC "x" THEN REFL_TAC;
        REPEAT STRIP_TAC THEN ASM_REWRITE_TAC[]]);;


let INTERIOR_DISJOINT = prove_thm (`INTERIOR_DISJOINT`,
  "!(s:(*)set) (x:*).
     ~((x IN INTERIOR s) /\ (x IN COMP(INTERIOR s)))",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[COMP;IN_DIFF;DE_MORGAN_THM;IN_UNIV]
    THEN REWRITE_TAC[ONCE_REWRITE_RULE[DISJ_SYM]
       (SPEC "(x IN (INTERIOR s)):bool"  EXCLUDED_MIDDLE)]);;


let INTERIOR_IDEM = prove_thm (`INTERIOR_IDEM`,
  "!(s:(*)set). INTERIOR s = s",
    GEN_TAC THEN REWRITE_TAC[EXTENSION;INTERIOR]
    THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
    THEN REWRITE_TAC[OPEN;OPEN_BALL;SUBSET_DEF]
    THEN GEN_TAC THEN EQ_TAC
    THENL [REPEAT STRIP_TAC THEN RES_TAC;
```

```
           REPEAT STRIP_TAC THEN EXISTS_TAC "s"
           THEN ASM_REWRITE_TAC[]
           THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
           THEN REWRITE_TAC[METRIC_lemma]]);;

let INTERIOR_EMPTY = prove_thm ('INTERIOR_EMPTY',
  "!(s:(*)set). (INTERIOR s = {}) ==> (s={})",
    CONV_TAC(ONCE_DEPTH_CONV SYM_CONV)
    THEN REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[]
    THEN MATCH_ACCEPT_TAC INTERIOR_IDEM);;

let INTERIOR_OPEN = prove_thm ('INTERIOR_OPEN',
  "!(s:(*)set). (s=INTERIOR s) = OPEN s",
    GEN_TAC
    THEN REWRITE_TAC[EXTENSION;INTERIOR;OPEN;
         SUBSET_DEF;OPEN_BALL]
    THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV) THEN EQ_TAC
    THENL [CONV_TAC(LEFT_IMP_FORALL_CONV) THEN EXISTS_TAC "x"
           THEN BOOL_CASES_TAC "(x IN s):bool"
   THENL [REWRITE_TAC[]
            THEN CONV_TAC LEFT_IMP_EXISTS_CONV
  THEN REPEAT STRIP_TAC THEN EXISTS_TAC "R"
  THEN EXISTS_TAC "d" THEN REPEAT STRIP_TAC
  THEN ASSUME_TAC DIST_REFL THEN RES_TAC
  THEN ASM_REWRITE_TAC[ZERO_LESS_EQ];
      REWRITE_TAC[]
  THEN CONV_TAC (DEPTH_CONV NOT_EXISTS_CONV)
  THEN REWRITE_TAC[DE_MORGAN_THM]
  THEN CONV_TAC(DEPTH_CONV LEFT_IMP_FORALL_CONV)
                THEN EXISTS_TAC "s" THEN STRIP_TAC
  THEN GEN_TAC THEN EXISTS_TAC "R"
                THEN EXISTS_TAC "d" THEN REPEAT STRIP_TAC
  THEN ASSUME_TAC DIST_REFL THEN RES_TAC
  THEN ASM_REWRITE_TAC[ZERO_LESS_EQ]];
            DISCH_TAC THEN GEN_TAC THEN EQ_TAC
        THENL [STRIP_TAC THEN EXISTS_TAC "s"
    THEN ASM_REWRITE_TAC[];
    CONV_TAC LEFT_IMP_EXISTS_CONV
    THEN REPEAT STRIP_TAC THEN RES_TAC]]);;


let INTERIOR_SUBSET = prove_thm (`INTERIOR_SUBSET`,
  "!(s:(*)set) (t:(*)set).
      (s SUBSET t) ==> ((INTERIOR s) SUBSET (INTERIOR t))",
     REPEAT GEN_TAC
     THEN REWRITE_TAC[INTERIOR;SUBSET_DEF]
     THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
     THEN REPEAT STRIP_TAC THEN EXISTS_TAC "t':(*)set"
     THEN ASM_REWRITE_TAC[] THEN REPEAT STRIP_TAC
     THEN RES_TAC THEN RES_TAC);;


let EXISTS_OPEN = prove_thm (`EXISTS_OPEN`,
  "!(s:(*)set)(t:(*)set).
     (?(t':(*)set). OPEN t' /\ (!x'. x' IN t'
         ==> x' IN s \/ x' IN t))",
    REPEAT GEN_TAC THEN EXISTS_TAC "t"
    THEN REWRITE_TAC[OPEN;OPEN_BALL]
    THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
    THEN REWRITE_TAC[METRIC_lemma]
    THEN REPEAT STRIP_TAC THEN ASM_REWRITE_TAC[]);;


let INTERIOR_INTER = prove_thm (`INTERIOR_INTER`,
  "!(s:(*)set) (t:(*)set).
      INTERIOR(s INTER t) =
        ((INTERIOR s) INTER (INTERIOR t))",
    REWRITE_TAC[INTERIOR_IDEM]);;


let INTERIOR_UNION = prove_thm (`INTERIOR_UNION`,
  "!(s:(*)set) (t:(*)set).
     ((INTERIOR s) UNION (INTERIOR t))
         SUBSET INTERIOR(s UNION t)",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[INTERIOR;SUBSET_DEF;IN_UNION]
    THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
    THEN REPEAT STRIP_TAC THEN EXISTS_TAC "t':(*)set"
    THEN ASM_REWRITE_TAC[] THEN REPEAT STRIP_TAC
    THEN RES_TAC THEN ASM_REWRITE_TAC[]);;

let INTERIOR_CLOSURE = prove_thm ('INTERIOR_CLOSURE',
  "!(s:(*)set). (INTERIOR s) SUBSET (INTERIOR(CLOSURE s))",
    GEN_TAC
    THEN REWRITE_TAC[INTERIOR;SUBSET_DEF;CLOSURE;IN_UNION]
    THEN REWRITE_TAC[LIMIT;NEIGHBOUR;EXTENSION]
    THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
    THEN REPEAT STRIP_TAC THEN EXISTS_TAC "t"
    THEN ASM_REWRITE_TAC[] THEN REPEAT STRIP_TAC
    THEN RES_TAC THEN ASM_REWRITE_TAC[]);;

let INTERIOR_DIFF = prove_thm ('INTERIOR_DIFF',
  "!(s:(*)set) (t:(*)set).
    INTERIOR(s DIFF t) = ((INTERIOR s) DIFF (INTERIOR t))",
    REWRITE_TAC[INTERIOR_IDEM]);;

%<-------------------------------------------------------
   Boundary definitions and theorems
 ------------------------------------------------------->%

let BOUNDARY = new_definition ('BOUNDARY_DEF',
  "BOUNDARY (s:(*)set) =
    {x| (x IN CLOSURE s) /\ (x IN CLOSURE(COMP s))}");;

let ON = new_definition ('ON_DEF',
  "ON (x,(s:(*)set)) = x IN BOUNDARY s");;
```

```
let BOUNDARY_lemma = prove_thm (`BOUNDARY_lemma`,
  "!(s:(*)set).
       BOUNDARY s = ((CLOSURE s) INTER (CLOSURE(COMP s)))",
    GEN_TAC THEN REWRITE_TAC[EXTENSION;IN_INTER;BOUNDARY]
    THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
    THEN REPEAT GEN_TAC THEN REFL_TAC);;

let BOUNDARY_CLOSED = prove_thm (`BOUNDARY_CLOSED`,
  "!(s:(*)set). CLOSED(BOUNDARY s)",
    REWRITE_TAC[CLOSED;BOUNDARY;COMP;OPEN;OPEN_BALL;CLOSURE]
    THEN REWRITE_TAC[IN_DIFF;IN_UNION;IN_UNIV]
    THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
    THEN REWRITE_TAC[LIMIT;NEIGHBOUR;IN_DIFF;
         IN_UNIV;EXTENSION]
    THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
    THEN REPEAT STRIP_TAC THEN EXISTS_TAC "R"
    THEN EXISTS_TAC "d" THEN REPEAT STRIP_TAC
    THEN ASSUME_TAC DIST_REFL THEN RES_TAC
    THEN ASM_REWRITE_TAC[ZERO_LESS_EQ]);;

let BOUNDARY_COMP = prove_thm (`BOUNDARY_COMP`,
  "!(s:(*)set). BOUNDARY s = BOUNDARY(COMP s)",
    GEN_TAC THEN REWRITE_TAC[EXTENSION;BOUNDARY;COMP_IDEM]
    THEN ONCE_LEFT_TO_RIGHT_DEPTH_FIRST_REWRITE_TAC[CONJ_SYM]
    THEN GEN_TAC THEN REFL_TAC);;

let CLOSURE_BOUNDED = prove_thm (`CLOSURE_BOUNDED`,
  "!(s:(*)set). CLOSURE s = (s UNION BOUNDARY s)",
    GEN_TAC
    THEN REWRITE_TAC[EXTENSION;CLOSURE;IN_UNION;BOUNDARY]
    THEN REWRITE_TAC[LIMIT;COMP;IN_DIFF;IN_UNIV]
    THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
    THEN GEN_TAC THEN EQ_TAC
  THENL [% Subgoal 1.0 %REPEAT STRIP_TAC
     THENL [% 1.1 % ASM_REWRITE_TAC[];
        % 1.2 % ONCE_ASM_REWRITE_TAC[]
THEN REWRITE_TAC[NEIGHBOUR;EXTENSION]
THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
THEN BOOL_CASES_TAC "(x IN s):bool"
THENL [% 1.2.1 % REWRITE_TAC[]
   THEN CONV_TAC LEFT_OR_EXISTS_CONV
   THEN EXISTS_TAC "x" THEN ONCE_REWRITE_TAC[EQ_SYM]
   THEN REWRITE_TAC[];
   % 1.2.2 % REWRITE_TAC[] THEN REPEAT GEN_TAC
   THEN EQ_TAC THEN REPEAT STRIP_TAC
   THENL [% 1.2.2.1 % ASM_REWRITE_TAC[];
   % 1.2.2.2 % EXISTS_TAC "y':(*)"
         THEN ASM_REWRITE_TAC[];
   % 1.2.2.3 % EXISTS_TAC "y':(*)"
         THEN ASM_REWRITE_TAC[];
   % 1.2.2.4 % ASM_REWRITE_TAC[];
   % 1.2.2.5 % EXISTS_TAC "y':(*)"
         THEN ASM_REWRITE_TAC[]]]];
         % Subgoal 2.0 % REPEAT STRIP_TAC
         THENL [% 2.1 % ASM_REWRITE_TAC[];
 % 2.2 % POP_ASSUM (\th. ALL_TAC)
         THEN ASM_REWRITE_TAC[];
 % 2.3 % ONCE_ASM_REWRITE_TAC[] THEN REWRITE_TAC[];
 % 2.4 % ONCE_ASM_REWRITE_TAC[]
          THEN REWRITE_TAC[NEIGHBOUR;EXTENSION]
    THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
    THEN REPEAT GEN_TAC THEN EQ_TAC
    THENL [% 2.4.1 % REPEAT STRIP_TAC
      THENL [ASM_REWRITE_TAC[];
             EXISTS_TAC "y':(*)" THEN ASM_REWRITE_TAC[];
   EXISTS_TAC "y':(*)" THEN ASM_REWRITE_TAC[]];
   % 2.4.2 % REPEAT STRIP_TAC
```

```
          THENL [ASM_REWRITE_TAC[];
                   EXISTS_TAC "y':(*)"
                   THEN ASM_REWRITE_TAC[]]];
              % 2.5 % ONCE_ASM_REWRITE_TAC[]
            THEN REWRITE_TAC[NEIGHBOUR;EXTENSION]
      THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
      THEN BOOL_CASES_TAC "(x IN s):bool"
      THENL [% 2.5.1 % REWRITE_TAC[]
        THEN CONV_TAC LEFT_OR_EXISTS_CONV
           THEN EXISTS_TAC "x"
           THEN ONCE_REWRITE_TAC[EQ_SYM] THEN REWRITE_TAC[];
           % 2.5.2 % REWRITE_TAC[]
           THEN REPEAT GEN_TAC THEN EQ_TAC
           THENL [% 2.5.2.1 % REPEAT STRIP_TAC
   THENL [ASM_REWRITE_TAC[];
               EXISTS_TAC "y':(*)" THEN ASM_REWRITE_TAC[];
 EXISTS_TAC "y':(*)" THEN ASM_REWRITE_TAC[]];
   % 2.5.2.2 % REPEAT STRIP_TAC
     THENL [ASM_REWRITE_TAC[];EXISTS_TAC "y':(*)"
               THEN ASM_REWRITE_TAC[]]]]]]);;


let CLOSURE_IDEM = prove_thm (`CLOSURE_IDEM`,
  "!(s:(*)set). (CLOSURE s) = s",
    GEN_TAC
    THEN REWRITE_TAC[EXTENSION;CLOSURE;LIMIT;IN_UNION]
    THEN REWRITE_TAC[NEIGHBOUR;EXTENSION]
    THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
    THEN GEN_TAC THEN EQ_TAC
    THENL [STRIP_TAC THEN POP_ASSUM MP_TAC
           THEN CONV_TAC LEFT_IMP_FORALL_CONV
           THEN EXISTS_TAC "s" THEN REPEAT STRIP_TAC
           THEN ASSUM_LIST(
             \t.ASSUME_TAC(REWRITE_RULE[(el 1 t)](el 2 t)))
           THEN POP_ASSUM MATCH_ACCEPT_TAC;
          REPEAT STRIP_TAC THEN ASM_REWRITE_TAC[]]);;

let CLOSURE_INTER = prove_thm (`CLOSURE_INTER`,
  "!(s:(*)set)(t:(*)set).
    CLOSURE(s INTER t) = ((CLOSURE s) INTER (CLOSURE t))",
    REWRITE_TAC[CLOSURE_IDEM]);;

let CLOSURE_lemma1 = prove_thm (`CLOSURE_lemma1`,
  "!(s:(*)set).
     CLOSURE s = ((INTERIOR s) UNION (BOUNDARY s))",
    REWRITE_TAC[BOUNDARY_lemma;INTERIOR_IDEM;
          CLOSURE_IDEM;COMP_DISJOINT;UNION_EMPTY]);;

let INTERIOR_COMP = prove_thm (`INTERIOR_COMP`,
  "!(s:(*)set). INTERIOR s = COMP(CLOSURE(COMP s))",
     GEN_TAC THEN ASSUME_TAC INTERIOR_IDEM
     THEN POP_ASSUM MP_TAC
     THEN CONV_TAC(ONCE_DEPTH_CONV SYM_CONV)
     THEN DISCH_TAC
     THEN POP_ASSUM(\th.ONCE_ASM_REWRITE_TAC[th])
     THEN ONCE_REWRITE_TAC[
        (SPEC "(COMP s):(*)set" CLOSURE_IDEM)]
     THEN REWRITE_TAC[COMP_IDEM;INTERIOR_IDEM]);;

let COMP_INTERIOR = prove_thm (`COMP_INTERIOR`,
  "!(s:(*)set). COMP(INTERIOR s) = CLOSURE(COMP s)",
    GEN_TAC THEN ASSUME_TAC(SPEC "s:(*)set" INTERIOR_COMP)
    THEN ASSUME_TAC
        (SPEC "(CLOSURE(COMP s)):(*)set" COMP_IDEM)
    THEN ASSUME_TAC(SPECL ["((INTERIOR s):(*)set)";
        "((COMP(CLOSURE(COMP s))):(*)set)"] COMP_EQ)
    THEN RES_TAC THEN ASSUM_LIST(\th. ASSUME_TAC(
         ONCE_REWRITE_RULE[(el 3 th)](el 1 th)))
    THEN POP_ASSUM MATCH_ACCEPT_TAC);;
```

```
let COMP_CLOSURE = prove_thm (`COMP_CLOSURE`,
  "!(s:(*)set). COMP(CLOSURE s) = INTERIOR(COMP s)",
    GEN_TAC THEN ASSUME_TAC(REWRITE_RULE[(COMP_IDEM)]
         (SPEC"(COMP s):(*)set" COMP_INTERIOR))
    THEN ASSUME_TAC
         (SPECL ["((COMP(INTERIOR(COMP s))):(*)set)";
         "((CLOSURE s):(*)set)"] COMP_EQ)
    THEN RES_TAC THEN ASSUM_LIST(\th.ASM_REWRITE_TAC[
         REWRITE_RULE[(COMP_IDEM)] (el 1 th)]));;

let CLOSURE_DISJOINT = prove_thm (`CLOSURE_DISJOINT`,
  "!(s:(*)set). DISJOINT (INTERIOR s) (BOUNDARY s)",
    REWRITE_TAC[INTERIOR_IDEM;BOUNDARY_lemma;CLOSURE_IDEM]
    THEN REWRITE_TAC[COMP_DISJOINT;
            DISJOINT_DEF;INTER_EMPTY]);;

let UNIV_COMPOSIT = prove_thm (`UNIV_COMPOSIT`,
  "!(s:(*)set) (t:(*)set).
     (INTERIOR s) UNION (BOUNDARY s) UNION
         (INTERIOR(COMP s)) = UNIV",
    REWRITE_TAC[INTERIOR_IDEM;BOUNDARY_lemma;CLOSURE_IDEM]
    THEN REWRITE_TAC[COMP_DISJOINT;UNION_EMPTY;COMP_UNIV]
    THEN REWRITE_TAC[EXTENSION;IN_UNIV]);;

let BOUNDARY_UNION_SUBSET = prove_thm
       (`BOUNDARY_UNION_SUBSET`,
  "!(s:(*)set) (t:(*)set).
    BOUNDARY(s UNION t) SUBSET ((BOUNDARY s)
         UNION (BOUNDARY t))",
    REWRITE_TAC[BOUNDARY_lemma;CLOSURE_IDEM;COMP_DISJOINT]
    THEN REWRITE_TAC[UNION_EMPTY;SUBSET_DEF]);;

let BOUNDARY_INTER_SUBSET = prove_thm
```

```
      ('BOUNDARY_INTER_SUBSET',
  "!(s:(*)set) (t:(*)set).
    BOUNDARY(s INTER t) SUBSET ((BOUNDARY s)
          UNION (BOUNDARY t))",
    REWRITE_TAC[BOUNDARY_lemma;CLOSURE_IDEM;COMP_DISJOINT]
    THEN REWRITE_TAC[UNION_EMPTY;SUBSET_DEF]);;

let BOUNDARY_DIFF_SUBSET = prove_thm ('BOUNDARY_DIFF_SUBSET',
  "!(s:(*)set) (t:(*)set).
    BOUNDARY(s DIFF t) SUBSET ((BOUNDARY s)
          UNION (BOUNDARY t))",
    REWRITE_TAC[BOUNDARY_lemma;CLOSURE_IDEM;COMP_DISJOINT]
    THEN REWRITE_TAC[UNION_EMPTY;SUBSET_DEF]);;

let INTERIOR_UNION_SUBSET = prove_thm
        ('INTERIOR_UNION_SUBSET',
  "!(s:(*)set) (t:(*)set).
    INTERIOR(s UNION t) SUBSET
      ((INTERIOR s) UNION (INTERIOR t) UNION
          ((BOUNDARY s) INTER (BOUNDARY t)))",
    REWRITE_TAC[INTERIOR_IDEM;BOUNDARY_lemma;CLOSURE_IDEM]
    THEN REWRITE_TAC[COMP_DISJOINT;INTER_EMPTY;
          UNION_EMPTY;SUBSET_DEF]);;

let BOUNDARY_UNION = prove_thm ('BOUNDARY_UNION',
  "!(s:(*)set) (t:(*)set).
    BOUNDARY(s UNION t) =
      (BOUNDARY s INTER (INTERIOR(COMP t))) UNION
      (INTERIOR(COMP s) INTER BOUNDARY t) UNION
      ((BOUNDARY s) INTER (BOUNDARY t) INTER
           CLOSURE(COMP s INTER COMP t))",
    REWRITE_TAC[BOUNDARY_lemma;CLOSURE_IDEM;INTERIOR_IDEM]
    THEN REWRITE_TAC[COMP_DISJOINT;
         INTER_EMPTY;UNION_EMPTY]);;
```

```
let BOUNDARY_INTER = prove_thm (`BOUNDARY_INTER`,
  "!(s:(*)set) (t:(*)set).
    BOUNDARY(s INTER t) =
      ((BOUNDARY s) INTER (INTERIOR t)) UNION
      ((INTERIOR s) INTER (BOUNDARY t)) UNION
      ((BOUNDARY s) INTER (BOUNDARY t) INTER
        CLOSURE(s INTER t))",
    REWRITE_TAC[INTERIOR_IDEM;BOUNDARY_lemma;
          CLOSURE_IDEM;COMP_DISJOINT]
    THEN REWRITE_TAC[UNION_EMPTY;INTER_EMPTY]);;

let BOUNDARY_DIFF = prove_thm (`BOUNDARY_DIFF`,
  "!(s:(*)set) (t:(*)set).
    BOUNDARY(s DIFF t) =
      ((BOUNDARY s) INTER (INTERIOR(COMP t))) UNION
      ((INTERIOR s) INTER (BOUNDARY t)) UNION
      ((BOUNDARY s) INTER (BOUNDARY t) INTER
         CLOSURE(s INTER (COMP t)))",
    REWRITE_TAC[INTERIOR_IDEM;BOUNDARY_lemma;
      CLOSURE_IDEM;COMP_DISJOINT]
    THEN REWRITE_TAC[UNION_EMPTY;INTER_EMPTY]);;

let CLOSED_BOUNDARY_INTER1 = prove_thm
        (`CLOSED_BOUNDARY_INTER1`,
  "!(s:(*)set) (t:(*)set).
    (CLOSED s) /\ (CLOSED t) ==>
     (BOUNDARY(s INTER t) =
      (((BOUNDARY s) INTER (INTERIOR t)) UNION
      ((INTERIOR s) INTER (BOUNDARY t)) UNION
      ((BOUNDARY s) INTER (BOUNDARY t))))",
    REWRITE_TAC[INTERIOR_IDEM;BOUNDARY_lemma;CLOSURE_IDEM]
    THEN REWRITE_TAC[COMP_DISJOINT;
        INTER_EMPTY;UNION_EMPTY]);;
```

```
let CLOSED_BOUNDARY_INTER2 = prove_thm
        (`CLOSED_BOUNDARY_INTER2`,
  "!(s:(*)set) (t:(*)set).
    (CLOSED s) /\ (CLOSED t) ==>
      (BOUNDARY(s INTER t) =
       (s INTER BOUNDARY t) UNION ((BOUNDARY s) INTER t))",
    REWRITE_TAC[BOUNDARY_lemma;CLOSURE_IDEM]
    THEN REWRITE_TAC[COMP_DISJOINT;
          INTER_EMPTY;UNION_EMPTY]);;

%<-------------------------------------------------------
 Dimensional property of boundary definitions and theorems
 ------------------------------------------------------->%
let THIN = new_definition (`THIN_DEF`,
  "THIN (s:(*)set) = (INTERIOR(CLOSURE s)={})");;

let THIN_SUBSET = prove_thm (`THIN_SUBSET`,
  "!(s:(*)set) (t:(*)set).
      (THIN s /\ (t SUBSET s)) ==> THIN t",
    REWRITE_TAC[THIN;INTERIOR_IDEM;CLOSURE_IDEM]
    THEN REPEAT STRIP_TAC
    THEN ASSUM_LIST(
      \th. ASSUME_TAC(REWRITE_RULE[(el 2 th)](el 1 th)))
    THEN ASSUM_LIST(\th. ONCE_ASM_REWRITE_TAC[
      REWRITE_RULE[SUBSET_EMPTY](el 1 th)])
    THEN REFL_TAC);;

let THINBOUND = new_definition (`THINBOUND_DEF`,
  "THINBOUND (s:(*)set) = THIN(BOUNDARY s)");;

let THINBOUND_CLOSED = prove_thm (`THINBOUND_CLOSED`,
  "!(s:(*)set). CLOSED s ==> THINBOUND s",
    GEN_TAC
    THEN REWRITE_TAC[THINBOUND;BOUNDARY_lemma;
        THIN;CLOSURE_IDEM]
    THEN REWRITE_TAC[COMP_DISJOINT] THEN STRIP_TAC
    THEN REWRITE_TAC[INTERIOR;EXTENSION;NOT_IN_EMPTY]
    THEN CONV_TAC(DEPTH_CONV SET_SPEC_CONV)
    THEN REWRITE_TAC[SUBSET_EMPTY]
    THEN CONV_TAC(DEPTH_CONV NOT_EXISTS_CONV)
    THEN REPEAT STRIP_TAC
    THEN ASSUM_LIST(
        \th.ASSUME_TAC(REWRITE_RULE[(el 2 th)](el 1 th)))
    THEN ASSUME_TAC NOT_IN_EMPTY THEN RES_TAC);;

let THINBOUND_OPEN = prove_thm (`THINBOUND_OPEN`,
  "!(s:(*)set). OPEN s ==> THINBOUND s",
    REWRITE_TAC[THINBOUND;THIN;CLOSURE_IDEM;INTERIOR_IDEM]
    THEN REWRITE_TAC[BOUNDARY_lemma;
        CLOSURE_IDEM;COMP_DISJOINT]);;

let THIN_UNION = prove_thm (`THIN_UNION`,
  "!(s:(*)set) (t:(*)set). THIN s /\ THIN t
      ==> THIN(s UNION t)",
    REPEAT GEN_TAC THEN REWRITE_TAC[THIN;INTERIOR_IDEM]
    THEN REWRITE_TAC[CLOSURE_IDEM]
    THEN REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[UNION_EMPTY]);;

let THIN_INTER = prove_thm (`THIN_INTER`,
  "!(s:(*)set) (t:(*)set).
      THIN s /\ THIN t ==> THIN(s INTER t)",
    REWRITE_TAC[THIN;CLOSURE_IDEM;INTERIOR_IDEM]
    THEN REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[INTER_EMPTY]);;

let THIN_DIFF = prove_thm (`THIN_DIFF`,
  "!(s:(*)set) (t:(*)set).
       THIN s /\ THIN t ==> THIN(s DIFF t)",
    REWRITE_TAC[THIN;CLOSURE_IDEM;INTERIOR_IDEM]
    THEN REPEAT STRIP_TAC
    THEN ASM_REWRITE_TAC[DIFF_EMPTY]);;

let THINBOUND_UNION_SUBSET1 = prove_thm
       ('THINBOUND_UNION_SUBSET1',
  "!(s:(*)set) (t:(*)set). THINBOUND s /\ THINBOUND t ==>
      ((INTERIOR(s UNION t)) SUBSET
           CLOSURE((INTERIOR s) UNION (INTERIOR t)))",
    REWRITE_TAC[THINBOUND;THIN;BOUNDARY_lemma;
        CLOSURE_IDEM;INTERIOR_IDEM]
    THEN REWRITE_TAC[COMP_DISJOINT;SUBSET_DEF]);;

let THINBOUND_UNION_SUBSET2 = prove_thm
      ('THINBOUND_UNION_SUBSET2',
  "!(s:(*)set) (t:(*)set). THINBOUND s /\ THINBOUND t ==>
     (CLOSURE(INTERIOR(s UNION t)) =
         CLOSURE((INTERIOR s) UNION (INTERIOR t)))",
    REWRITE_TAC[THINBOUND;THIN;BOUNDARY_lemma;
        CLOSURE_IDEM;INTERIOR_IDEM]);;

let THINBOUND_INTER_SUBSET1 = prove_thm
      ('THINBOUND_INTER_SUBSET1',
  "!(s:(*)set) (t:(*)set). THINBOUND s /\ THINBOUND t ==>
      (INTERIOR((CLOSURE s) INTER (CLOSURE t)) SUBSET
           CLOSURE(s INTER t))",
    REWRITE_TAC[THINBOUND;THIN;BOUNDARY_lemma;
        CLOSURE_IDEM;INTERIOR_IDEM]
    THEN REWRITE_TAC[COMP_DISJOINT;SUBSET_DEF]);;

let THINBOUND_INTER_SUBSET2 = prove_thm
       ('THINBOUND_INTER_SUBSET2',
  "!(s:(*)set) (t:(*)set). THINBOUND s /\ THINBOUND t ==>
    (INTERIOR(CLOSURE(s INTER t)) =
        INTERIOR((CLOSURE s) INTER (CLOSURE t)))",
    REWRITE_TAC[THINBOUND;THIN;BOUNDARY_lemma;
         CLOSURE_IDEM;INTERIOR_IDEM]
    THEN REWRITE_TAC[COMP_DISJOINT;SUBSET_DEF]);;

%<------------------------------------------------------
   Regularity definitions and theorems
 ------------------------------------------------------>%
let REGULAR = new_definition (`REGULAR_DEF`,
  "REGULAR (s:(*)set) = CLOSURE(INTERIOR s)");;

let REGULARISED = new_definition (`REGULARISED_DEF`,
  "REGULARISED (s:(*)set) = (s = REGULAR s)");;

let REGULAR_CLOSED = prove_thm (`REGULAR_CLOSED`,
  "!(s:(*)set). CLOSED(REGULAR s)",
    GEN_TAC
    THEN REWRITE_TAC[REGULAR;CLOSURE_CLOSED]
    THEN REWRITE_TAC[SPEC "(CLOSURE s):(*)set"
         CLOSURE_IDEM]);;

let THIN_REGULAR = prove_thm (`THIN_REGULAR`,
  "!(s:(*)set). THINBOUND(REGULAR s)",
    REWRITE_TAC[THINBOUND;THIN;REGULAR;BOUNDARY_lemma]
    THEN REWRITE_TAC[CLOSURE_IDEM;INTERIOR_IDEM;
         COMP_DISJOINT]);;

let REGULAR_EMPTY = prove_thm (`REGULAR_EMPTY`,
  "!(s:(*)set).
       REGULARISED s /\ (INTERIOR s ={}) ==> (s={})",
    REPEAT STRIP_TAC THEN ASSUME_TAC INTERIOR_EMPTY
    THEN RES_TAC);;
```

```
let REGULAR_IDEM = prove_thm (`REGULAR_IDEM`,
  "!(s:(*)set). REGULAR s = s",
    GEN_TAC
    THEN REWRITE_TAC[REGULAR;CLOSURE_IDEM;INTERIOR_IDEM]);;

let REGULAR_REFL = prove_thm (`REGULAR_REFL`,
  "!(s:(*)set). REGULARISED(REGULAR s)",
    GEN_TAC THEN REWRITE_TAC[REGULARISED;REGULAR_IDEM]);;

let IMP_LEFT = prove_thm (`IMP_LEFT`,
  " !a b c. (a ==> b /\ c) ==> (a ==> b)",
    REPEAT GEN_TAC THEN BOOL_CASES_TAC "a:bool"
    THEN REWRITE_TAC[AND1_THM]);;

let IN_IMP1 = prove_thm (`IN_IMP1`,
  "!(s:(*)set)(t:(*)set)(t':(*)set).
     (!x. x IN t' ==> x IN s /\ x IN t)
        ==> (!x. x IN t' ==> x IN s)",
    REPEAT GEN_TAC THEN DISCH_TAC THEN REPEAT STRIP_TAC
    THEN RES_TAC);;

let IN_IMP2 = prove_thm (`IN_IMP2`,
  "!(s:(*)set)(t:(*)set)(t':(*)set).
     (!x. x IN t' ==> x IN s /\ x IN t)
        ==> (!x. x IN t' ==> x IN t)",
    REPEAT GEN_TAC THEN DISCH_TAC THEN REPEAT STRIP_TAC
    THEN RES_TAC);;

let REGULAR_INTER = prove_thm (`REGULAR_INTER`,
  "!(s:(*)set)(t:(*)set).
    REGULAR(s INTER t) = ((REGULAR s) INTER (REGULAR t))",
    REWRITE_TAC[REGULAR;CLOSURE_IDEM;INTERIOR_IDEM]
    THEN REFL_TAC);;
```

```
%<----------------------------------------------------

   Regularised-operator definitions and theorems

 ---------------------------------------------------->%

let REG_UNION = new_infix_definition (`REG_UNION_DEF`,
  "$REG_UNION (s:(*)set) (t:(*)set) = REGULAR(s UNION t)");;


let REG_INTER = new_infix_definition (`REG_INTER_DEF`,
  "$REG_INTER (s:(*)set) (t:(*)set) = REGULAR(s INTER t)");;


let REG_DIFF = new_infix_definition (`REG_DIFF_DEF`,
  "$REG_DIFF (s:(*)set) (t:(*)set) = REGULAR(s DIFF t)");;


let REG_COMP = new_definition (`REG_COMP_DEF`,
  "$REG_COMP (s:(*)set) = REGULAR(COMP s)");;


let INTERIOR_REG_INTER = prove_thm (`INTERIOR_REG_INTER`,
  "!(s:(*)set) (t:(*)set).
    INTERIOR(s REG_INTER t) =
         INTERIOR((REGULAR s) INTER (REGULAR t))",
    REWRITE_TAC[REGULAR;REG_INTER;
         CLOSURE_IDEM;INTERIOR_IDEM]);;


let REGULAR_REG_INTER = prove_thm (`REGULAR_REG_INTER`,
  "!(s:(*)set) (t:(*)set).
    s REG_INTER t = ((REGULAR s) REG_INTER (REGULAR t))",
    REWRITE_TAC[REGULAR;REG_INTER;
         CLOSURE_IDEM;INTERIOR_IDEM]);;


let REGULAR_UNION = prove_thm (`REGULAR_UNION`,
  "!(s:(*)set) (t:(*)set).
    REGULAR(s UNION t) = ((REGULAR s) UNION (REGULAR t))",
    REWRITE_TAC[REGULAR;INTERIOR_IDEM;CLOSURE_IDEM]);;
```

```
let REGULARISED_REG_INTER = prove_thm
       (`REGULARISED_REG_INTER`,
  "!(s:(*)set) (t:(*)set).
    ((REGULARISED s) /\ (REGULARISED t)) ==>
      (INTERIOR(s REG_INTER t) =
         ((INTERIOR s) INTER (INTERIOR t)))",
    REWRITE_TAC[REG_INTER;REGULAR;
         INTERIOR_IDEM;CLOSURE_IDEM]);;


let REGULARISED_REG_UNION = prove_thm
       (`REGULARISED_REG_UNION`,
  "!(s:(*)set) (t:(*)set).
    ((REGULARISED s) /\ (REGULARISED t)) ==>
      ((s REG_UNION t) = (s UNION t))",
    REWRITE_TAC[REGULARISED;REG_UNION;REGULAR;
         INTERIOR_IDEM;CLOSURE_IDEM]);;


let REGULARISED_REG_DIFF = prove_thm (`REGULARISED_REG_DIFF`,
  "!(s:(*)set) (t:(*)set).
    ((REGULARISED s) /\ (REGULARISED t)) ==>
     ((s REG_DIFF t) = (s REG_INTER (REG_COMP t)))",
    REWRITE_TAC[REGULARISED;REG_INTER;
        REGULAR;REG_COMP;REG_DIFF]
    THEN REWRITE_TAC[INTERIOR_IDEM;CLOSURE_IDEM;EXTENSION]
    THEN REWRITE_TAC[IN_DIFF;IN_INTER;COMP;
          IN_DIFF;IN_UNIV]);;


let REG_COMP_lemma = prove_thm (`REG_COMP_lemma`,
  "!(s:(*)set). CLOSED s ==>
         ((REG_COMP s) = CLOSURE(COMP s))",
    REWRITE_TAC[REG_COMP;REG_DIFF;CLOSURE_IDEM;COMP;REGULAR]
    THEN REWRITE_TAC[CLOSURE_IDEM;INTERIOR_IDEM]);;


let REG_DIFF_lemma = prove_thm (`REG_DIFF_lemma`,
  "!(s:(*)set) (t:(*)set).
        (s REG_DIFF t) = (s REG_INTER (COMP t))",
    REWRITE_TAC[REG_DIFF;REG_INTER;COMP;REGULAR;EXTENSION]
    THEN REWRITE_TAC[CLOSURE_IDEM;INTERIOR_IDEM;
         IN_INTER;IN_DIFF;IN_UNIV]);;


%<-----------------------------------------------------
   Recursion formulae for interior, boundary, complement
 ------------------------------------------------------>%

let COMP_REG_UNION = prove_thm ('COMP_REG_UNION',
  "!(s:(*)set) (t:(*)set).
      COMP(s REG_UNION t) = ((COMP s) INTER (COMP t))",
    REWRITE_TAC[REG_UNION;COMP;REGULAR]
    THEN REWRITE_TAC[CLOSURE_IDEM;INTERIOR_IDEM;EXTENSION]
    THEN REWRITE_TAC[IN_DIFF;IN_INTER;IN_UNION;IN_UNIV]
    THEN REWRITE_TAC[DE_MORGAN_THM]);;

let INTERIOR_REG_COMP = prove_thm ('INTERIOR_REG_COMP',
  "!(s:(*)set) (t:(*)set). INTERIOR(REG_COMP s)=COMP s",
    REWRITE_TAC[REG_COMP;INTERIOR_IDEM;CLOSURE_IDEM]
    THEN REWRITE_TAC[REGULAR;INTERIOR_IDEM;
         CLOSURE_IDEM;COMP]);;

let BOUNDARY_REG_COMP = prove_thm ('BOUNDARY_REG_COMP',
  "!(s:(*)set) (t:(*)set).
        BOUNDARY(REG_COMP s) = BOUNDARY s",
    REWRITE_TAC[REGULARISED;REG_COMP;REGULAR;BOUNDARY_lemma]
    THEN REWRITE_TAC[CLOSURE_IDEM;INTERIOR_IDEM;COMP_IDEM]
    THEN ONCE_LEFT_TO_RIGHT_DEPTH_FIRST_REWRITE_TAC
         [INTER_COMM] THEN GEN_TAC THEN REFL_TAC);;

let COMP_REG_COMP = prove_thm ('COMP_REG_COMP',
  "!(s:(*)set) (t:(*)set). COMP(REG_COMP s) = INTERIOR s",
    REWRITE_TAC[REG_COMP;REGULAR]
   THEN REWRITE_TAC[INTERIOR_IDEM;CLOSURE_IDEM;COMP_IDEM]);;

let INTERIOR_REG_DIFF = prove_thm (`INTERIOR_REG_DIFF`,
  "!(s:(*)set) (t:(*)set).
      INTERIOR(s REG_DIFF t) =
          ((INTERIOR s) INTER (COMP t))",
   REWRITE_TAC[REG_DIFF;REGULAR;INTERIOR_IDEM]
   THEN REWRITE_TAC[CLOSURE_IDEM;COMP;EXTENSION]
   THEN REWRITE_TAC[IN_INTER;IN_DIFF;IN_UNIV]);;

let BOUNDARY_REG_UNION = prove_thm (`BOUNDARY_REG_UNION`,
  "!(s:(*)set) (t:(*)set).
     BOUNDARY(s REG_UNION t) =
      (((BOUNDARY s) INTER (COMP t)) UNION
       ((COMP t) INTER (BOUNDARY t)) UNION
       ((BOUNDARY s) INTER (BOUNDARY t) INTER
          (CLOSURE((COMP s) INTER (COMP t)))))",
     REWRITE_TAC[REG_UNION;REGULAR]
     THEN REWRITE_TAC[BOUNDARY_lemma;CLOSURE_IDEM;
          INTERIOR_IDEM;COMP_DISJOINT;
          INTER_EMPTY;UNION_EMPTY]);;

let BOUNDARY_REG_INTER = prove_thm (`BOUNDARY_REG_INTER`,
  "!(s:(*)set) (t:(*)set).
      BOUNDARY(s REG_INTER t) =
       (((BOUNDARY s) INTER (INTERIOR t)) UNION
        ((BOUNDARY t) INTER (INTERIOR s)) UNION
        ((BOUNDARY s) INTER (BOUNDARY t) INTER
           (CLOSURE((INTERIOR s) INTER (CLOSURE t)))))",
     REWRITE_TAC[REG_INTER;REGULAR;BOUNDARY_lemma]
     THEN REWRITE_TAC[CLOSURE_IDEM;
         INTERIOR_IDEM;COMP_DISJOINT]
     THEN REWRITE_TAC[INTER_EMPTY;UNION_EMPTY]);;
```

```
let BOUNDARY_REG_DIFF = prove_thm (`BOUNDARY_REG_DIFF`,
  "!(s:(*)set) (t:(*)set).
      BOUNDARY(s REG_DIFF t) =
       (((CLOSURE s) INTER (BOUNDARY(COMP t))) UNION
        ((INTERIOR s) INTER (BOUNDARY t)) UNION
        ((BOUNDARY s) INTER (BOUNDARY t) INTER
           (CLOSURE((INTERIOR s) INTER (COMP t)))))",
     REWRITE_TAC[REG_DIFF;REGULAR;BOUNDARY_lemma]
     THEN REWRITE_TAC[CLOSURE_IDEM;
        INTERIOR_IDEM;COMP_DISJOINT]
     THEN REWRITE_TAC[INTER_EMPTY;UNION_EMPTY]);;


%<-----------------------------------------------------
   Membership classifications
 ------------------------------------------------------>%

let in = new_infix_definition (`in_DEF`,
  "in (t:(*)set) (s:(*)set) = t REG_INTER (INTERIOR s)");;


let on = new_infix_definition (`on_DEF`,
  "on (t:(*)set) (s:(*)set) = t REG_INTER (BOUNDARY s)");;


let out = new_infix_definition (`out_DEF`,
  "out (t:(*)set) (s:(*)set) = t REG_INTER (COMP s)");;


let MEMBER_lemma = prove_thm (`MEMBER_lemma`,
  "!(s:(*)set) (t:(*)set).
     t = (t in s) UNION (t on s) UNION (t out s)",
    REWRITE_TAC[in;on;out;REG_INTER;REGULAR;
           CLOSURE_IDEM;INTERIOR_IDEM]
    THEN REWRITE_TAC[BOUNDARY_lemma;
           CLOSURE_IDEM;COMP_DISJOINT]
    THEN REWRITE_TAC[INTER_EMPTY;UNION_EMPTY;EXTENSION]
    THEN REWRITE_TAC[IN_UNION;IN_INTER]
    THEN REPEAT GEN_TAC THEN EQ_TAC
    THENL [STRIP_TAC THEN ASM_REWRITE_TAC[COMP]
           THEN REWRITE_TAC[IN_DIFF;IN_UNIV;EXCLUDED_MIDDLE];
           REPEAT STRIP_TAC]);;

let MEMBER_DISJOINT = prove_thm (`MEMBER_DISJOINT`,
  "!(s:(*)set) (t:(*)set).
     ((t in s) REG_INTER (t on s) = {}) /\
     ((t on s) REG_INTER (t out s) = {}) /\
     ((t in s) REG_INTER (t out s) = {})",
    REWRITE_TAC[in;on;out;REG_INTER;REGULAR;
        CLOSURE_IDEM;INTERIOR_IDEM]
    THEN REWRITE_TAC[BOUNDARY_lemma;
        CLOSURE_IDEM;COMP_DISJOINT]
    THEN REWRITE_TAC[INTER_EMPTY;EXTENSION;
        IN_INTER;COMP;IN_DIFF]
    THEN REWRITE_TAC[IN_UNIV;NOT_IN_EMPTY]
    THEN REPEAT GEN_TAC THEN REWRITE_TAC[DE_MORGAN_THM]
    THEN BOOL_CASES_TAC "((x:*) IN (t:(*)set)):bool"
    THENL [ONCE_REWRITE_TAC[DISJ_SYM]
      THEN REWRITE_TAC[SPEC "(x IN s):bool" EXCLUDED_MIDDLE];
      REWRITE_TAC[]]);;

let IN_SUBSET = prove_thm (`IN_SUBSET`,
  "!(s:(*)set) (t:(*)set).
      (t in s) SUBSET REGULAR(INTERIOR s)",
    REWRITE_TAC[in;REG_INTER;REGULAR;
        INTERIOR_IDEM;CLOSURE_IDEM]
    THEN REWRITE_TAC[SUBSET_DEF;IN_INTER]
    THEN REPEAT STRIP_TAC);;

let ON_SUBSET = prove_thm (`ON_SUBSET`,
  "!(s:(*)set) (t:(*)set).
      (t on s) SUBSET REGULAR(BOUNDARY s)",
    REWRITE_TAC[on;REG_INTER;REGULAR;
```

```
       BOUNDARY_lemma;CLOSURE_IDEM]
    THEN REWRITE_TAC[INTERIOR_IDEM;SUBSET_DEF;IN_INTER]
    THEN REPEAT STRIP_TAC THEN ASM_REWRITE_TAC[]);;


let OUT_SUBSET = prove_thm ('OUT_SUBSET',
  "!(s:(*)set) (t:(*)set). (t out s) SUBSET REGULAR(COMP s)",
    REWRITE_TAC[out;REG_INTER;REGULAR;
        CLOSURE_IDEM;INTERIOR_IDEM]
    THEN REWRITE_TAC[SUBSET_DEF;IN_INTER]
    THEN REPEAT STRIP_TAC);;


let MEMBER_INTER = prove_thm ('MEMBER_INTER',
  "!(s:(*)set) (s1:(*)set) (s2:(*)set) (t:(*)set).
    (s = s1 REG_INTER s2) ==>
        ((t in s) = (t in s1) REG_INTER (t in s2))",
    REWRITE_TAC[in;REG_INTER;REGULAR;
        CLOSURE_IDEM;INTERIOR_IDEM]
    THEN REWRITE_TAC[EXTENSION;IN_INTER]
    THEN REPEAT STRIP_TAC THEN ASM_REWRITE_TAC[]
    THEN EQ_TAC THEN REPEAT STRIP_TAC
    THEN PURE_ASM_REWRITE_TAC[]);;

%<------------------------------------------------------------
   Description of points in 3-D coordinates
 ------------------------------------------------------------>%


let Point_Axiom = define_type 'Point_Axiom'
  'point = POINT num num num';;


let X = new_recursive_definition
  false Point_Axiom 'X_DEF'
    "X (POINT x y z) = x";;


let Y = new_recursive_definition
```

```
  false Point_Axiom `Y_DEF`
    "Y (POINT x y z) = y";;

let Z = new_recursive_definition
  false Point_Axiom `Z_DEF`
    "Z (POINT x y z) = z";;

%<-----------------------------------------------------------
    A type object contains two subsets of points
    One is set of points on the interior and the other
    is set of points on the boundary
    The constraints on an object type is that the baoundary
    is finite and the interior and boundary are disjoint.
 ----------------------------------------------------------->%
let Object_Axiom = define_type `Object_Axiom`
  `object = OBject (point)set (point)set`;;

let b = new_recursive_definition
  false Object_Axiom `b_DEF`
    "b(OBject interior bound) = bound";;

let i = new_recursive_definition
  false Object_Axiom `i_DEF`
    "i(OBject interior bound) = interior";;

let Closure = new_definition (`Closure_DEF`,
  "Closure A = b(A) UNION i(A)");;

let HAS_CARD = new_definition (`HAS_CARD_DEF`,
  "HAS_CARD (A:(*)set) n = CARD A = n");;

%< closure of an object is both upper and lower bounded >%
let OBJECT = new_definition (`OBJECT_DEF`,
  "OBJECT A = CLOSED(Closure A) /\ ~(b A = EMPTY) /\
    (?n. HAS_CARD (Closure A) n)");;


let OBJ = new_definition (`OBJ_DEF`,
  "OBJ A = OBject (i A) (b A)");;


let obj_constructor_11 = prove_constructors_one_one
           Object_Axiom;;


%<---------------------------------------------------------
   Some facts about objects
---------------------------------------------------------->%
%< existency theorem >%
let EXISTS_OBJECT = prove_thm (`EXISTS_OBJECT`,
  "?A. OBJECT A",
    REWRITE_TAC[OBJECT;Closure]
    THEN EXISTS_TAC "OBJECT
        (EMPTY:(point)set) ((x:point) INSERT EMPTY)"
    THEN REWRITE_TAC[b;i;IN_UNION]
    THEN STRIP_TAC
    THENL [REWRITE_TAC[CLOSED];
           CONV_TAC NOT_EXISTS_CONV
           THEN REWRITE_TAC[NOT_IN_EMPTY]]);;


%< Physically exists>%
let FINITE_OBJECT = prove_thm (`FINITE_OBJECT`,
  "!A. OBJECT A ==> ~(Closure A = {})",
    GEN_TAC THEN REWRITE_TAC[OBJECT;Closure;DISJOINT_DEF]
    THEN STRIP_TAC THEN REWRITE_TAC[EMPTY_UNION]
    THEN ASM_REWRITE_TAC[]);;


%< uniqueness theorem >%
let UNIQUE_OBJECT = prove_thm (`UNIQUE_OBJECT`,
  "!M N. OBJECT M /\ OBJECT N
    ==> (b(M) = b(N)) /\ (i(M) = i(N)) ==> (OBJ M = OBJ N)",
    REPEAT GEN_TAC
    THEN REWRITE_TAC[OBJECT;OBJ;obj_constructor_11]
    THEN REPEAT STRIP_TAC THEN ASM_REWRITE_TAC[]);;

%< two disjoint primitives have no common member >%
let DISJOINT_MEMBER = prove_thm ('DISJOINT_MEMBER',
  "!A B. DISJOINT (Closure A) (Closure B) ==>
    (i A INTER i B = EMPTY) /\ (b A INTER i B = EMPTY) /\
      (i A INTER b B = EMPTY) /\ (b A INTER b B = EMPTY)",
        REPEAT GEN_TAC
        THEN REWRITE_TAC[Closure;DISJOINT_DEF;
            UNION_OVER_INTER;EMPTY_UNION]
        THEN ONCE_REWRITE_TAC[INTER_COMM]
        THEN REWRITE_TAC[UNION_OVER_INTER;EMPTY_UNION]
        THEN DISCH_TAC THEN ASM_REWRITE_TAC[]);;

%<----------------------------------------------------------
   Some basic definitions about objects in 3-D Euclidean
   space
 ----------------------------------------------------------->%

%< A dot by definition is an object with no interior
   and a boundary of one member >%
let DOT = new_definition ('DOT_DEF',
  "DOT D = (OBJECT D) /\
    (CARD(b D)=1) /\ (i D = EMPTY)");;

%< A curve by definition is an object with finite
   interior and finite boundary >%
let CURVE = new_definition ('CURVE_DEF',
  "CURVE C = (OBJECT C) /\
    ~(i C = EMPTY) /\ FINITE(b C)");;

%< A curve is open if there exists exactly two
   boundary points >%
```

```
let OPEN_CURVE = new_definition ('OPEN_CURVE',
  "OPEN_CURVE C = CURVE C /\ (HAS_CARD(b C)1)");;


%< A closed curve has finite interior with no boundary >%
let CLOSED_CURVE = new_definition ('CLOSED_CURVE_DEF',
  "CLOSED_CURVE C = CURVE C /\ (b C=EMPTY)");;


%< A surface by definition is an object which has
   finite interior and a boundary of at least one
   closed curve >%
let SURFACE = new_definition ('SURFACE_DEF',
  "SURFACE s = (OBJECT s) /\ ~(i s = EMPTY) /\
    (?C. CLOSED_CURVE C /\ Closure C SUBSET (b s))");;


%< A primitive is an simplest constructionall object
   This is an abstract definition : a primitive has
   finite interior and its boundary has at least one
   closed surface>%
let PRIMITIVE = new_definition ('PRIMITIVE_DEF',
  "PRIMITIVE p = (OBJECT p) /\
    (?s. SURFACE s /\ Closure s SUBSET
        (b p) /\ ~(i p = EMPTY))");;


let HOLLOWED = new_definition ('HOLLOWED_DEF',
  "HOLLOWED k = (PRIMITIVE k) /\
    (!s1 s2. SURFACE s1 /\ SURFACE s2
      ==> DISJOINT (Closure s1) (Closure s2))");;


%<---------------------------------------------------------
   Some basic geometrical facts
 ---------------------------------------------------------->%
%< boundary and interior points are distinct >%
let NON_OVERLAP_ELEM = prove_thm ('NON_OVERLAP_ELEM',
  "!A. OBJECT A ==> (i A INTER b A = EMPTY)",
    REWRITE_TAC[OBJECT;DISJOINT_DEF]
    THEN REPEAT STRIP_TAC);;

%< dot and a curve is also distinct >%
let DOT_NOT_CURVE = prove_thm ('DOT_NOT_CURVE',
  "!D. DOT D ==> ~(CURVE D)",
    REWRITE_TAC[DOT;CURVE] THEN REPEAT STRIP_TAC
    THEN RES_TAC);;

%< dot and surface is also distinct >%
let DOT_NOT_SURFACE = prove_thm ('DOT_NOT_SURFACE',
  "!D. DOT D ==> ~(SURFACE D)",
    REWRITE_TAC[DOT;SURFACE] THEN REPEAT STRIP_TAC
    THEN RES_TAC);;

%<--------------------------------------------------------
   Some basic properties of a primitive
 --------------------------------------------------------->%
%< a primitive has at least one point i.e a dot >%
let NON_EMPTY = prove_thm ('NON_EMPTY',
  "!p. PRIMITIVE p ==> ?a. a IN (Closure p)",
    GEN_TAC THEN REWRITE_TAC[PRIMITIVE;OBJECT]
    THEN DISCH_TAC
    THEN REPEAT(POP_ASSUM(ASSUME_TAC o(\th. CONJUNCT1 th)))
    THEN ASSUME_TAC(INST_TYPE[":point",":*"]
          MEMBER_NOT_EMPTY) THEN RES_TAC
    THEN REWRITE_TAC[Closure;IN_UNION]
    THEN EXISTS_TAC "x:point" THEN ASM_REWRITE_TAC[]);;

%< an element of an object is enclosed >%
let ENCLOSED = prove_thm ('ENCLOSED',
  "!p r. (p IN Closure r) = (p IN (b r)) \/ (p IN (i r))",
    REPEAT STRIP_TAC THEN REWRITE_TAC[Closure]
    THEN EQ_TAC
    THENL[STRIP_TAC THEN POP_ASSUM(ASSUME_TAC o(\th.
         REWRITE_RULE[Closure;IN_UNION] th))
         THEN POP_ASSUM MATCH_ACCEPT_TAC;
         DISCH_TAC THEN REWRITE_TAC[IN_UNION]
         THEN ASM_REWRITE_TAC[]]);;

%< primitive is definable in a finite space >%
let BOUNDED = prove_thm ('BOUNDED',
  "!P. PRIMITIVE P ==> ~(Closure P={})",
    GEN_TAC
    THEN REWRITE_TAC[PRIMITIVE;OBJECT;Closure;EMPTY_UNION]
    THEN REPEAT STRIP_TAC THEN RES_TAC);;

%< a primitive is connected if any two points on its
   boundary can be joined by a curve                 >%
let CONNECTED = prove_thm ('CONNECTED',
  "!P.?C. PRIMITIVE P /\
    CURVE C /\ (Closure C SUBSET Closure P) ==>
      (!A B. (A IN Closure C) /\ (B IN Closure C) ==>
        (A IN Closure P) /\ (B IN Closure P))",
    REPEAT STRIP_TAC THEN EXISTS_TAC "C:object"
    THEN DISCH_TAC
    THEN POP_ASSUM (ASSUME_TAC o(\th. CONJUNCT2 th))
    THEN POP_ASSUM (ASSUME_TAC o(\th. CONJUNCT2 th))
    THEN ASSUME_TAC(INST_TYPE[":point",":*"] SUBSET_DEF)
    THEN REPEAT STRIP_TAC THEN RES_TAC);;

%< dot is not a primitive >%
let DOT_NOT_PRIM = prove_thm ('DOT_NOT_PRIM',
  "!D. DOT D ==> ~(PRIMITIVE D)",
    REWRITE_TAC[PRIMITIVE;DOT]
    THEN REPEAT STRIP_TAC THEN RES_TAC);;

%< inclusion property - a point is either inside an object
```

```
   or on its boundary or in its interior>%
let POINT_INCLUSION = prove_thm ('POINT_INCLUSION',
  "!a p. ~(a IN Closure p) \/ a IN (b p) \/ a IN (i p)",
    REPEAT GEN_TAC THEN REWRITE_TAC[Closure;IN_UNION]
    THEN MATCH_ACCEPT_TAC (ONCE_REWRITE_RULE[DISJ_SYM]
         EXCLUDED_MIDDLE));;

%<-------------------------------------------------------------
   Analytical descriptions of object
 ------------------------------------------------------------>%
%< A plane in 3-D representation >%
let PLANE = new_definition ('PLANE_DEF',
  "PLANE p = (SURFACE p) /\
      (!h. (h IN Closure(p)) ==>
          ?k l m n. k*X(h)+l*Y(h)+m*Z(h) = n)");;

%< Boundary of a region in space. Any plane divides the space
   into 2 half-spaces: one further from the origin
   called FAR_SPACE and one nearer, called NEAR_SPACE
   which both includes the surface itself>%

let FAR_SPACE = new_definition ('FAR_SPACE_DEF',
  "FAR_SPACE f(k,l,m,n) = (SURFACE f) /\
      (!x. (x IN Closure(f)) ==>
          (k*X(x)+l*Y(x)+m*Z(x)) >= n)");;

let NEAR_SPACE = new_definition ('NEAR_SPACE_DEF',
  "NEAR_SPACE f(k,l,m,n) = (SURFACE f) /\
      (!x. (x IN Closure(f)) ==>
          (k*X(x)+l*Y(x)+m*Z(x)) <= n)");;

%< A solid block is defined by its 6 surfaces. These are
   left,right,back, front, bottom and top respectively
   Parameters are : c is centre of block
```

```
                 l is half length (along x-axis)
                 w is half width (along y-axis)
                 h is half height (along z-axis)>%

let BLOCK = new_definition ('BLOCK_DEF',
  "BLOCK (dx,dy,dz)(l,w,h) =
     {p | ?e. PRIMITIVE e /\ p IN e /\
      (FAR_SPACE (b e)(1,0,0,dx) /\
      (NEAR_SPACE (b e)(1,0,0,l+dx)) /\
      (FAR_SPACE (b e)(0,1,0,dy)) /\
      (NEAR_SPACE (b e)(0,1,0,w+dy)) /\
      (FAR_SPACE (b e)(0,0,1,dz) /\
      (NEAR_SPACE (b e)(0,0,1,h+dz}))))))))");;

%< Square of a number >%
let SQR = new_definition ('SQR_DEF',
  "SQR(x) = x*x");;


%< A solid cylindrical object is defined by
     m(x,y,z) : centre of cylinder
           r  : radius
           h  : half height (along axis)
  The cylinder is an open region which is
  defined as an infinite tube >%

let CYLINDER_X = new_definition ('CYLINDER_X_DEF',
  "CYLINDER_X (dx,dy,dz)(r,h) =
     {p | ?c. PRIMITIVE c /\ p IN c /\
         (NEAR_SPACE (b c)(1,0,0,dx) /\
         (FAR_SPACE (b c)(1,0,0,h+dx) /\
         (SQR(Y(p)+dy)+SQR(Z(p)+dz)<=SQR(r))))");;

let CYLINDER_Y = new_definition ('CYLINDER_Y_DEF',
  "CYLINDER_Y (dx,dy,dz)(r,h) =
    {p | ?c. PRIMITIVE c /\ p IN c /\
         (NEAR_SPACE (b c)(0,1,0,dy) /\
         (FAR_SPACE (b c)(0,1,0,h+dy) /\
         (SQR(X(p)+dx)+SQR(Z(p)+dz)<=SQR(r))))");;

let CYLINDER_Z = new_definition ('CYLINDER_Z_DEF',
  "CYLINDER_Z (dx,dy,dz)(r,h) =
    {p | ?c. PRIMITIVE c /\ p IN c /\
         (NEAR_SPACE (b c)(0,0,1,dz) /\
         (FAR_SPACE (b c)(0,0,1,h+dz) /\
         (SQR(X(p)+dx)+SQR(Y(p)+dy)<=SQR(r))))");;

%<------------------------------------------------------------
     Build function to combined list of objects
 ------------------------------------------------------------>%

let BUILD = new_list_recursive_definition ('BUILD_DEF',
  "BUILD [] = \f. EMPTY /\
   BUILD (CONS hd tl) = \f. hd f BUILD [tl]");;

let NULL_DETECT = new_definition ('NULL_DETECT_DEF',
  "NULL_DETECT s = BUILD s REG_INTER = {}");;

%<------------------------------------------------------------
      Computation of overall volume
 ------------------------------------------------------------>%

let within = new_infix_definition ('within_DEF',
  "within n (k,p) = k<=p /\ k<=n /\ n<=p");;

% Overall volume of rectangular block %
let OVERALL_B = new_definition ('OVERALL_B_DEF',
  "OVERALL_B (dx,dy,dz)(l,w,h) =
    {p | X(p) within (dx,l+dx) /\
         Y(p) within (dy,w+dy) /\
         Z(p) within (dz,h+dz)");;
```

```
% Overall volume of an X-cylinder %
let OVERALL_CX = new_definition (`OVERALL_CX_DEF`,
  "OVERALL_CX (dx,dy,dz)(r,h) =
       OVERALL_B (dx,dy,dz)(h,2*r,2*r)");;

% Overall volume of an Y-cylinder %
let OVERALL_CY = new_definition (`OVERALL_CY_DEF`,
  "OVERALL_CY (dx,dy,dz)(r,h) =
       OVERALL_B (dx,dy,dz)(2*r,h,2*r)");;

% Overall volume of an Z-cylinder %
let OVERALL_CZ = new_definition (`OVERALL_CZ_DEF`,
  "OVERALL_CZ (dx,dy,dz)(r,h) =
       OVERALL_B (dx,dy,dz)(2*r,2*r,h)");;

let well_order1 = prove_thm (`well_order1`,
  "!n a b c d. (n within (a,b) /\ n within (c,d))
     ==> (a<=c /\ d<=b) => n within (a,b) | n within (c,d)",
      REPEAT GEN_TAC THEN REWRITE_TAC[within]
      THEN COND_CASES_TAC
      THENL [REPEAT STRIP_TAC THEN ASM_REWRITE_TAC[];
             REPEAT STRIP_TAC THEN ASM_REWRITE_TAC[]]);;

let well_order2 = prove_thm (`well_order2`,
  "!n a b c d. (n within (a,b) \/ n within (c,d))
     ==> (a<=c /\ d<=b) => n within (a,b) | n within (a,d)",
      REPEAT GEN_TAC THEN REWRITE_TAC[within]
      THEN COND_CASES_TAC
      THENL [REPEAT STRIP_TAC
             THENL [ASM_REWRITE_TAC[];ASM_REWRITE_TAC[];
                    ASM_REWRITE_TAC[];
             ASSUME_TAC(
              SPECL ["a:num";"c:num";"d:num"] LESS_EQ_TRANS)
```

```
            THEN RES_TAC
            THEN ASSUM_LIST (\th. ASSUME_TAC(
               REWRITE_RULE[(el 6 th)] (el 1 th)))
            THEN ASSUME_TAC(
               SPECL ["a:num";"d:num";"b:num"] LESS_EQ_TRANS)
            THEN RES_TAC
            THEN ASSUM_LIST (\th.
               ACCEPT_TAC(REWRITE_RULE[(el 9 th)] (el 1 th)));
            ASSUME_TAC(
               SPECL ["a:num";"c:num";"n:num"] LESS_EQ_TRANS)
            THEN RES_TAC
            THEN ASSUM_LIST (\th.
               ACCEPT_TAC(REWRITE_RULE[(el 6 th)] (el 1 th)));
            ASSUME_TAC(
               SPECL ["n:num";"d:num";"b:num"] LESS_EQ_TRANS)
            THEN RES_TAC
            THEN ASSUM_LIST (\th.
               ACCEPT_TAC(REWRITE_RULE[(el 6 th)] (el 1 th)))];
        POP_ASSUM MATCH_ACCEPT_TAC];;

let well_order3 = prove_thm ('well_order3',
  "!n a b c d. (n within (a,b) /\ ~(n within (c,d))) ==>
     (c<=b /\ d<=b) => (n within (a,c) /\ n within (d,b)) |
     (c<=b /\ b<=d) => n within (a,c) | n within (a,b)",
       REPEAT GEN_TAC THEN REWRITE_TAC[within]
       THEN REPEAT STRIP_TAC THEN COND_CASES_TAC
       THENL [ASM_REWRITE_TAC[] THEN REPEAT STRIP_TAC
          THEN ASSUME_TAC(
               SPECL ["a:num";"d:num";"b:num"] LESS_EQ_TRANS)
            THEN RES_TAC
            THEN ASSUM_LIST (\th.
               ACCEPT_TAC(REWRITE_RULE[(el 9 th)] (el 1 th)));
            ASSUME_TAC(
               SPECL ["a:num";"c:num";"n:num"] LESS_EQ_TRANS)
            THEN RES_TAC;
          POP_ASSUM MATCH_ACCEPT_TAC]);;

close_theory();;
```

345 350 358 362 375