Design Oriented Verification and Evaluation: The DOVE Project

Tony Cant, Brendan Mahony and Jim McCarthy

Information Networks Division
Information Sciences Laboratory

DSTO–TR–1349

ABSTRACT

DOVE is a graphical tool for modelling and reasoning about state machine designs for critical systems. This report summarizes its technical development, and incorporates the user manual.

RELEASE LIMITATION

Approved For Public Release

DOVE Release Notice

DOVE is being released on the understanding that it will be used for research purposes only, and that any reference to it acknowledges DSTO. The code can be freely copied and distributed as long as any copies include this release information. Requests for commercial use of DOVE should be made directly to DSTO.

The Commonwealth of Australia owns the copyright in DOVE and any other intellectual property in or relating to DOVE and you will have a non-exclusive licence only to use DOVE for your research.

As a user of DOVE, you accept the sole risk of interpreting and applying DOVE, which is provided as a general research tool only and should not be regarded or relied on as any form of professional advice or service in relation to your research. The Commonwealth of Australia gives no warranty, other than may reasonably be implied by legislation, that DOVE is free of fault or is the correct or sole research tool to be used for your research, and will not be liable to you or any third party for any loss or damage howsoever caused whether due to your negligence or otherwise, arising from the use of DOVE or reliance on the information contained in DOVE.

Published by

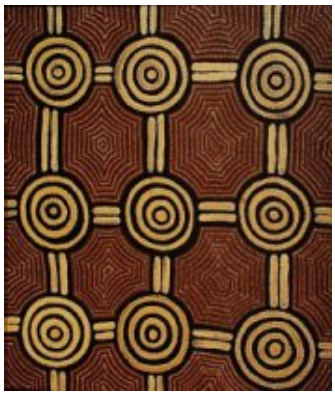
DSTO Information Sciences Laboratory
PO Box 1500
Edinburgh, South Australia, Australia 5111

Telephone: *(08) 8259 5555*
Facsimile: *(08) 8259 6567*

AR No. AR 012–457
October, 2002

APPROVED FOR PUBLIC RELEASE

"Bush Potato", by Glory Ngarla
Reproduced by kind permission of
Dacou Aboriginal Gallery



Design Oriented Verification and Evaluation: The DOVE Project

Executive Summary

Critical systems, such as safety-critical and security-critical systems, require the highest levels of engineering assurance. The achievement of such high levels of assurance must be based on the adoption of the most rigorous available analysis techniques. The most sophisticated assurance techniques make use of formal languages with strictly-defined semantics and are referred to as *formal methods*. international standards (for example UK Defence Standards 00-55 [13] and 00-56 [14]; ITSEC [3]; and Def(AUST) Standard 5679 [1]) mandate the rigorous analysis of system designs through the application of formal methods techniques. Thus, there is a clear need for tool support which will facilitate such analyses through the various stages of the design process.

The tool for Design Oriented Verification and Evaluation (DOVE), was developed by the Defence Science and Technology Organisation (DSTO) in Australia to meet this challenge. It provides a simple, but powerful, means of applying formal modelling and verification techniques to the design of safety- and security-critical systems. The DOVE tool is unusual among formal methods tools in that the overriding aim has been to provide a minimalist approach to the application of formal methods. In other words, it will aid the different participants of the design process without significant disruption of their standard practice. Broadly speaking, it is expected that the users will fall into three related categories: *system designers*, who have expertise in particular application domains and wish to use the tool to model and explore systems of interest; *evaluators*, with a general, rather than detailed, understanding of the application and proof structure, who must be convinced of the soundness of the design (e.g., by animation) and proofs (e.g., by replay) without constructing any themselves; and *verifiers*, who have expertise in the use of the prover, but not necessarily of a given application.

The DOVE tool has three main components: a *graph editor*, for constructing state machine diagrams; an *animator*, for exploring symbolic execution of the machine; and a *prover*, for verifying critical properties of the machine. This separation provides a degree of flexibility in the design process which is essential when dealing with large-scale engineering problems.

DOVE adopts the ubiquitous state machine as its basic design model. State machines are familiar to most design oriented professions, particularly in the engineering and computer science

fields. They are an effective means of communicating system designs to a variety of people and are easy to represent and to manipulate with a graphical user interface.

This manual is written at three different levels. At one level it provides an analysis of the problem of tool support for formal design, and the formalisation of state machine modelling of critical systems. Below this is a straightforward introduction to the use of the DOVE tool, largely "tutorial" in nature, based around a specific example state machine. It allows the reader quickly to get into experimenting with state machine design. The lowest level provides a more complete description of the DOVE tool and some of its finer points, to be used more as a reference source or as background for the interested user.

DOVE has been designed by:

Tony Cant
Katherine Eastaughffe
Jim Grundy
Jim McCarthy
Brendan Mahony
Maris Ozols

Programming support from:

Tim Anstee, University of South Australia
Moira Clarke, Ebor Computing
Geoff Tench, CSC (Australia)

Contributions from:

Helen Daniel
Tony Dekker
Mark Klink
Chuchang Liu
John Yesberg

Correspondence pursuant to DOVE should be addressed to:

Trusted Computer Systems Group
Information Networks Division
Information Sciences Laboratory
Defence Science and Technology Organisation
PO Box 1500, Edinburgh
South Australia 5111
Email: dove@dsto.defence.gov.au

Authors

Tony Cant
TCS group

Tony Cant, Ph.D., is a Senior Research Scientist within the Trusted Computer Systems Group of DSTO. He leads a section that carries out research into high assurance methods and tools, focusing on approaches to machine-assisted reasoning about their critical properties. He also provides technical and policy advice to the Defence Materiel Organisation on safety management issues, and is the editor-in-chief of the Def (Aust) 5679 Standard entitled ”The Procurement of Computer-Based Safety Critical Systems”, published by the Land Engineering Agency

Brendan Mahony
TCS group

Brendan Mahony has actively investigated the application of formal mathematics to the development of trusted systems in a number of areas, including multi-level secure systems, security protocols, real-time, concurrent, and reactive systems since 1988. He was awarded a PhD in real-time systems verification from the University of Queensland in 1991 and has worked for the Trusted Computer Systems group at the DSTO since 1995.

Jim McCarthy
TCS group

Jim McCarthy came to the Trusted Computer Systems group in 1998, via a career in theoretical physics dating back to his Ph.D. at Rockefeller University in 1985. He is working as a Research Scientist to develop high assurance methods and tools, to model specific (typically infosec) critical systems, and to monitor progress in Quantum Computation.

Contents

Glossary xv

Chapter 1 Introduction **1**

1.1 Reading this report as a user manual 1

1.2 Design assurance 2

1.2.1 Modelling 2

1.2.2 Animation 2

1.2.3 Verification 3

1.2.4 Reviews and Checks 3

1.2.5 Stakeholders 3

1.3 The DOVE approach 3

1.3.1 Use of graphical interfaces 3

1.3.2 Use of existing design strategies 4

1.3.3 Use of generic software components 4

1.4 State machine design 4

1.4.1 The editor 6

1.4.2 The animator 7

1.4.3 The prover 7

Chapter 2 DOVE state machines **9**

2.1 System attributes 10

2.2 Transitions 11

2.3 Initialisation 12

2.4 State machine definitions 12

2.5 Executions 13

2.6 Animation 14

2.7 Properties 14

2.8 Verification 17

2.9 Scope of DOVE 19

Chapter 3 A first look at DOVE **21**
3.1 Starting DOVE . . . 21
3.2 DOVE files . . . 22
3.2.1 State machine graph file . . . 23
3.2.2 Noweb files . . . 23
3.2.3 Theory files . . . 23
3.2.4 Image file . . . 23
3.2.5 High-resolution documentation file . . . 24
3.3 DOVE tools . . . 24
3.3.1 Edit mode . . . 24
3.3.2 Animation mode . . . 24
3.3.3 Proof mode . . . 25

Chapter 4 Editing the state machine **27**
4.1 Graph editing on the canvas . . . 27
4.1.1 Graph layout . . . 28
4.1.2 Nodes . . . 28
4.1.3 Edges . . . 30
4.2 The menu bar of the DOVE state machine window . . . 31
4.2.1 The File menu . . . 31
4.2.2 The Edit menu . . . 32
4.2.3 The View menu . . . 32
4.2.4 The Definitions menu . . . 33
4.2.5 Other displays on the menu bar . . . 37
4.3 Transitions . . . 38
4.3.1 The Let declaration . . . 38
4.3.2 The Guard declaration . . . 38
4.3.3 The action list declaration . . . 39
4.3.4 Editing, deleting and renaming transitions . . . 39
4.4 Mandatory elements of state machine design . . . 40
4.4.1 The identifiers . . . 40
4.4.2 Rules for initialisation of the state machine . . . 41
4.4.3 Naming rules . . . 41

4.4.4 Declaration and type rules . . . 41
4.4.5 Check assignments . . . 41
4.5 Tutorial: construction of TrafficLights . . . 42
4.5.1 Topology . . . 42
4.5.2 Moving graph objects . . . 43
4.5.3 Labelling graph objects . . . 43
4.5.4 Machine definition . . . 44
4.5.4.1 Datatypes . . . 46
4.5.4.2 Constants . . . 47
4.5.4.3 Variables . . . 48
4.5.4.4 Initialisation . . . 48
4.5.4.5 Transitions . . . 48
4.5.5 Renaming a transition . . . 49
4.5.6 Saving and reloading the state machine . . . 49

Chapter 5 Animation 51

5.1 DOVE window display in animation mode . . . 51
5.1.1 The Watch Variable window . . . 52
5.1.2 The `Animator` window . . . 52
5.1.3 Path conditions in the `Animator` window . . . 53
5.2 Animation via the state machine graph . . . 54
5.2.1 Starting the animations . . . 54
5.2.2 Animation . . . 54
5.2.3 The Animation Controls . . . 55
5.2.4 Named Animations . . . 55
5.3 The Animator menu bar . . . 56
5.3.1 The File menu . . . 56
5.3.2 The Windows menu . . . 57
5.3.3 Exiting Animation Mode . . . 58
5.4 Tutorial: animation of TrafficLights . . . 58

Chapter 6 Managing Proofs **61**

6.1 Window display on entering proof mode 61

6.2 The Properties Manager window 61

6.2.1 `Property Status` reporting 62

6.2.2 The `Options` menu 63

6.2.3 Exiting the proof mode 64

6.3 Tutorial: example properties in TrafficLights 64

6.4 The Prover window 66

6.4.1 Frames and buttons 67

6.4.2 The menu bar 68

6.4.2.1 The `File` menu 68

6.4.2.2 The `Proof` menu 68

6.4.2.3 The `View` menu 69

6.4.2.4 The `Options` menu 69

6.4.2.5 The `Swap` menu 69

6.5 Proof management: ending, saving, loading and restarting proofs 69

6.6 The Theorem Browser window 71

6.6.1 Frames 71

6.6.2 The menu bar 71

6.6.3 Matching terms facility 72

6.7 Proof visualization 72

Chapter 7 Proof strategies and tactics **75**

7.1 Proof in DOVE/XIsabelle 76

7.1.1 Interactive proof tools 76

7.1.2 Theorems and inference 77

7.1.3 Proof-state and tactics 79

7.1.4 Temporal sequents 80

7.2 The DOVE proof strategy 80

7.2.1 The primary tactics of the DOVE proof strategy 81

7.2.2 Augmenting the basic strategy 84

7.3 The DOVE Tactics frame 85

7.3.1 Primary tactics 85

7.3.2 `Temporal Machine` tactics . . . 85
7.3.3 `Interactive` tactics . . . 86
7.3.4 `Configuration` tactics . . . 87
7.4 Introductory Tutorial: the DOVE proof strategy . . . 87
7.4.1 Induction . . . 88
7.4.2 Topology . . . 88
7.4.3 Back-substitution . . . 89
7.5 Advanced Tutorial: proof management in practice . . . 90
7.5.1 Intermediate lemma method . . . 91
7.5.2 `Add Invariant` method . . . 95
7.5.3 Using `MasterBlast` . . . 96
7.6 Methods used in the Advanced Tutorial . . . 96
7.6.1 Keeping invariants using `Add Invariant` . . . 97
7.6.2 Proof scripts . . . 97
7.6.3 A brief look at the `Basic Tactics` used . . . 99
7.6.4 Applying constant definitions via `Basic Tactics` . . . 101
References . . . 103

Appendices

A The TrafficLights state machine . . . 104
B Syntax of DOVE . . . 107
B.1 Transition definition . . . 107
B.2 Sequent . . . 107
C Rules of temporal logic . . . 110
C.1 Structural rules . . . 110
C.2 Rewriting equalities . . . 118
D State machine diagnostics . . . 123
D.1 Checks in compilation . . . 123
D.2 Diagnostic messages in compilation . . . 124
D.2.1 Initialisation checks . . . 124
D.2.2 Edge to transition checks . . . 124
D.2.3 Structure Checks . . . 125
D.2.4 Uncommitted data . . . 125

D.3 Parsing errors in definitions . . . 125
D.3.1 Type errors in defining type abbreviations and variables . . . 126
D.3.2 Errors in transition input . . . 126
D.3.3 Errors in property input . . . 127
E Dealing with errors . . . 129
E.1 General overview of error interaction . . . 129
E.2 Known bugs or design flaws in DOVE . . . 130
E.2.1 Edit mode . . . 130
E.2.2 Animation mode . . . 130
E.2.3 Proof mode . . . 131
F Troubleshooting DOVE . . . 133
G Questions and answers about the formal proof model . . . 134
G.1 What is a formal theory, anyway? . . . 134
Index . . . 135

Figures

1.1 DOVE processes block diagram 5
1.2 DOVE state machine structure 6
2.1 state machine graph for the `TrafficLights` example. 10
2.2 A possible execution path for the `TrafficLights` machine 13
3.1 DOVE state machine window at startup. 22
3.2 DOVE mode buttons 24
4.1 A DOVE state machine during node movement 29
4.2 The state machine segment initially. 44
4.3 The edited state machine segment. 45
4.4 The labelled state machine segment. 46
5.1 The Watch Variables window. 52
5.2 DOVE display after one step in an animation. 53
6.1 The Properties Manager window. 62
6.2 The edit property dialog window. 65
6.3 The prover window 67
7.1 The structure of a theorem (or proof state). 77
7.2 The assumption inference rule 78
7.3 The resolution inference rule 78
7.4 Tree Display after configuration tactics applied 93

Glossary

The following is a glossary of technical terms that are used in this User Manual. Most of the terms are defined in a DOVE-specific context.

Action: A program which changes the values of a *state machine's heap variables*.

Axiom: A *proposition* which is true a priori in a given *theory* – it makes up part of the definition of the theory.

Behaviour: A sequence of *configurations* exhibited during an execution of a *state machine*.

Configuration: The values of all of the observable attributes of a *state machine*, including the *control state*, the *last transition*, and the *memory*.

Discharging a subgoal: Proving that a *subgoal* is a *theorem* through the application of a single *tactic*.

Formal design: Expressing the state machine in the chosen *formal language*.

Formal language: A language which has strict rules of grammar and unambiguous semantics. DOVE has *formal languages* for describing *state machines* and their *properties*.

Formal proof: A procedure for deriving a *theorem*, using only the *axioms* and *inference rules* of a *theory*.

Goal: The original *proposition* to be proved.

Guard: A Boolean condition in the *state machine memory* which determines if a *transition* is enabled.

Heap variable: A named component of a *state machine's memory* describing attributes determined by the machine.

Inference rule: A function for deriving new *theorems* from known *theorems* in a given *theory*. In Isabelle all inference rules belong to the `Pure` theory.

Input variable: A named component of a *state machine's memory* which describes the environment which the *state machine* observes or makes use of during computation.

Lemma: A *theorem* which is only constructed to be used in the proof of another (more "important") *theorem*.

Memory: The collection of a *state machine's heap* and *input* variables.

Proof state: The list of currently unproved *subgoals* at a given stage in the proof construction. The *subgoals* are always constructed such that if they can be proved to be *theorems* then the original *goal* of the proof is also a *theorem*. A proof is complete when all the *subgoals* have been *discharged*.

Proof tool (Automatic): Computer software which attempts to construct a *formal proof* of a given *proposition*.

Proof tool (Interactive): Computer software which helps a human operator to construct a *formal proof* of a given *proposition*.

Property: A statement describing a collection of possible *behaviours* of a *state machine*. In DOVE, properties are expressed in *temporal logic*. The term *property* is overloaded to also refer to the *proposition* that the *behaviours* of a *state machine* satisfy a *property*.

Proposition: A truth valued statement in a formal language.

State: A node in a *state machine's* control graph.

State machine: A model of computation in which the control logic is described (and depicted) as a graph in which the nodes describe the possible control *states* and the arcs describe the allowed *transitions* for each *state*. DOVE *state machines* are augmented with a *memory*.

Subgoal: A *proposition* which appears as an intermediate step in the proof of a *goal*.

Tactic: A program used to transform the current *proof state*. The new *proof state* introduced by the *tactic* is always one from which the previous *proof state* can be logically inferred; that is. the previous *subgoals* can be constructed from the new *subgoals* and existing theorems using the *inference rules*. A tactic is neither a *theorem* nor an *inference rule*, but will make use of *theorems* and *inference rules*. Generally a *tactic* will work by transforming a single *subgoal*, but this is not mandatory.

Temporal logic: A *formal language* for expressing the *properties* of *state machines*. Mechanisms are provided for testing both the current *configuration* and past *configurations*, as well as the ordering of events.

Temporal sequent: A formal mechanism for stating *propositions* about the behaviour of a *state machine*. A *sequent* consists of a list of hypothesis *properties* and a target *property*, written in the form

```
H1, ..., Hn |- T
```

It states the *proposition* that if the *state machine* satisfies the hypothesis *properties* it also satisfies the target *property*.

Theorem: A *proposition* which can be derived from the *axioms* of a *theory* using the *inference rules*. In particular, the *axioms* themselves are *theorems*. The Isabelle *proof tool* has a reserved type called `thm`, and only *axioms* and the results of *inference rules* have this type. This ensures the integrity of the Isabelle reasoning system.

Theory: A collection of formal objects and the *axioms* which describe their meaning. Isabelle also maintains a database of *theorems* that have been proved in a theory.

Theory file: A file containing a program which Isabelle uses to construct a *theory*.

Transition: An arc in a *state machine's* control graph and its associated *guard* and *action*.

Chapter 1

Introduction

Broadly speaking, it is expected that DOVE users will fall into three related categories: *system designers*, who have expertise in particular application domains and wish to use the tool to model and explore systems of interest; *evaluators*, with a general, rather than detailed, understanding of the application and proof structure, who must be convinced of the soundness of the design (e.g., by animation) and proofs (e.g., by replay) without constructing any themselves; and *verifiers*, who have expertise in the use of the prover, but not necessarily of a given application.

The benefits to these users from using DOVE are flexibility, clarity, and ease-of-use. These derive from the underlying design of the tool, as will be discussed in detail below. The main points, briefly, are:

- DOVE clearly separates the activities of design modelling, design animation and design verification. This provides a degree of flexibility in the design process which is essential when dealing with large-scale engineering problems.
- DOVE adopts the ubiquitous state machine as its basic design model. State machines are familiar to most design oriented professions, particularly in the engineering and computer science fields. They are an effective means of communicating system designs to a variety of people and are easy to represent and to manipulate with a graphical user interface.
- DOVE provides an intelligent graphical interface which emphasises both simplicity of use, and automated support for design and analysis activities. This provides productivity benefits to both the novice and the experienced user.

It is also important to note that the use of state machines reduces the training effort required to start making effective use of the DOVE tool.

1.1 Reading this report as a user manual

As a user manual, this report is written at two different levels. At one level it is a straightforward introduction to the use of the DOVE tool, largely “tutorial” in nature, based around a specific example state machine. It allows the reader quickly to get into experimenting with state machine design. The other level provides a more complete description of the DOVE tool and some of its

finer points. It is expected that this latter thread would be used more as a reference source, or as background for the interested user. As many readers prefer simply to download a tool and use it, this section provides a guide to the introductory and tutorial aspects of the manual.

Firstly, the reader may want to peruse the remainder of this introduction, at least to gain confidence with the definitions and meanings of the various concepts in the design process. For a brief overview of DOVE's implementation of state machines, it is highly desirable to read Chapter2. It concentrates on an example, and is fairly "hands-on", giving the reader a firm grasp of what the tool is about.

The reader should then have a quick look at Chapter3, particularly Section 3.1, which rapidly discuss the necessary preliminaries to setting up a DOVE session. Moreover, the preamble to Section 3.2 has a useful discussion of "good practice" in file management while using the tool.

After these preliminaries the reader will be able to attempt the "tutorial sessions", which appear as the last sections in the remaining chapters; specifically, Sections4.5, 5.4, 6.3, 7.4 and 7.5. Most of these can be followed "blindly", but it is expected that reference to earlier material in those chapters will be useful during the sessions. The last tutorials, Sections7.4 and 7.5, are somewhat harder in that the proof strategy is incorporated in the discussion.

1.2 Design assurance

DOVE is primarily a tool for producing high assurance system designs. The reason for placing such a high importance on the level of design assurance of a system is that experience has shown that it is in this phase that extra assurance efforts can have the most effect. Design errors are relatively easy and inexpensive to correct if detected during the actual design phase, but when left undetected through to later stages of the development can become difficult and expensive to eradicate. In the extreme case, a design error can lead to the complete wastage of all subsequent development efforts. It is also significantly easier to detect errors in the design phase, because designs should be small, elegant, and easy to understand. As with most areas of human endeavour, the more time spent in planning and preparation, the smoother the execution.

The design assurance process, as embodied in the DOVE system, covers a range of issues which are worth discussing at this point.

1.2.1 Modelling

Modelling is the activity of discovering a simple description of some real-world system. Design modelling provides increased understanding of, and confidence in, a design. Depending on the degree of accuracy required, models may vary in sophistication from simple diagrams, to scaled physical models, to complex mathematical objects.

1.2.2 Animation

Animation is the process of exercising a design model to observe its behaviour. Animation may be used to determine the accuracy of a model or else to determine the fitness-for-purpose of

a design. The method of animating a model is dependent on the form of the model. Scale models are generally tested physically, while mathematical models may be subjected to formal reasoning or to computer simulation.

1.2.3 Verification

Verification is the activity of direct mathematical or logical proof that a model meets certain critical requirements. Verification may be either rigorous – that is, demonstrated by arguments that are convincing to human evaluators – or else formal, that is, demonstrated by arguments that have been or are amenable to being checked by computers. Although it can be expensive to carry out, design verification can be a very effective way of providing design assurance, as well as discovering design flaws.

1.2.4 Reviews and Checks

Reviews and checks are design analysis activities carried out by external parties. Subjecting a design to the scrutiny of an objective (or at least more objective) observer can add greatly to confidence in the adequacy and correctness of a design.

1.2.5 Stakeholders

Design assurance activities can involve a number of parties, with widely differing roles and capabilities. System designers are generally experts in the application domain of the system, but they may, in consequence, have a narrow range of mathematical and logical competencies. Verifiers will be familiar with a range of very general mathematical and logical techniques, but may not possess the specific skills necessary for successful design in a particular application domain. Evaluators and certifiers review the development process for compliance with appropriate standards and best practice. Although evaluators may not have detailed knowledge of either the application domain or formal proof techniques, it is the evaluators that the design assurance activities must convince.

1.3 The DOVE approach

An important factor in promoting more widespread use of formal methods in design assurance is the provision of tool support. The DOVE system provides tools for constructing, presenting and reasoning about formal design-models. The design and development of DOVE has been informed by three basic beliefs, outlined below, about the provision of tool support for the application of formal methods.

1.3.1 Use of graphical interfaces

The provision of a powerful graphical user interface is critical to the successful application of formal methods to large-scale designs. The power of a tool's user interface should not be seen

simply in terms of providing an attractive display and easy-to-use point-and-click control of tool functionality. It is important that the interface should also provide a powerful visual metaphor for the design model being analysed. A suitable visual metaphor makes the application of the tool more intuitive and provides vital feedback when using the tool. Such an interface enhances the appeal of the tool for novice users, reduces the effort required to achieve basic competency, and makes formal modelling less error-prone.

1.3.2 Use of existing design strategies

It is important that formal methods tools should not simply provide a powerful interface to formal modelling and analysis techniques. They should also integrate well with existing design methods. From a very pragmatic point of view, this is desirable because it facilitates acceptance of the tool amongst those involved in the design process. However, even from a purer design point of view it has the advantage of allowing the tool to be modest in scope. It allows the tool to be tightly focused on the job of providing suitable formal analysis support, without the need to address wider methodological issues.

1.3.3 Use of generic software components

As depicted in Figure 1.1, the DOVE prototype has primarily been built using three existing tools and/or languages: the Tcl/Tk script language [6], the functional programming language ML [7], and the interactive theorem-proving system Isabelle [9]. The user interface component and the graphical representation of the state machine are implemented in Tcl and Tk. The process communication language, Expect [5], is used to implement communication between Tcl/Tk and ML. All parsing of user input is carried directly in Isabelle, while proofs are carried out using XIsabelle [2], a graphical front-end to Isabelle.

1.4 State machine design

The *state machine* is a widely used system design model, forming an important component of many existing design processes and design support tools. State machines provide a powerful, flexible model of computation. They are able to treat a wide range of design issues at convenient levels of abstraction. They are readily amenable to the application of powerful automated tools, such as silicon compilers [10], as well as formal analysis techniques. Furthermore, when rendered in the form of transition graphs, state machines provide an important design visualisation tool, able to communicate effectively to a wide range of stakeholders.

The DOVE system adopts the state machine as the basic model of system design. This has the dual benefits of providing a basic design model which is easily rendered graphically and which allows the DOVE tool to be readily integrated into existing design practices.

The main features of DOVE state machines are displayed in Figure 1.2. The diagram presents a DOVE model of the operation of a set of traffic lights at a north-south/east-west road junction, and we may note some general features.

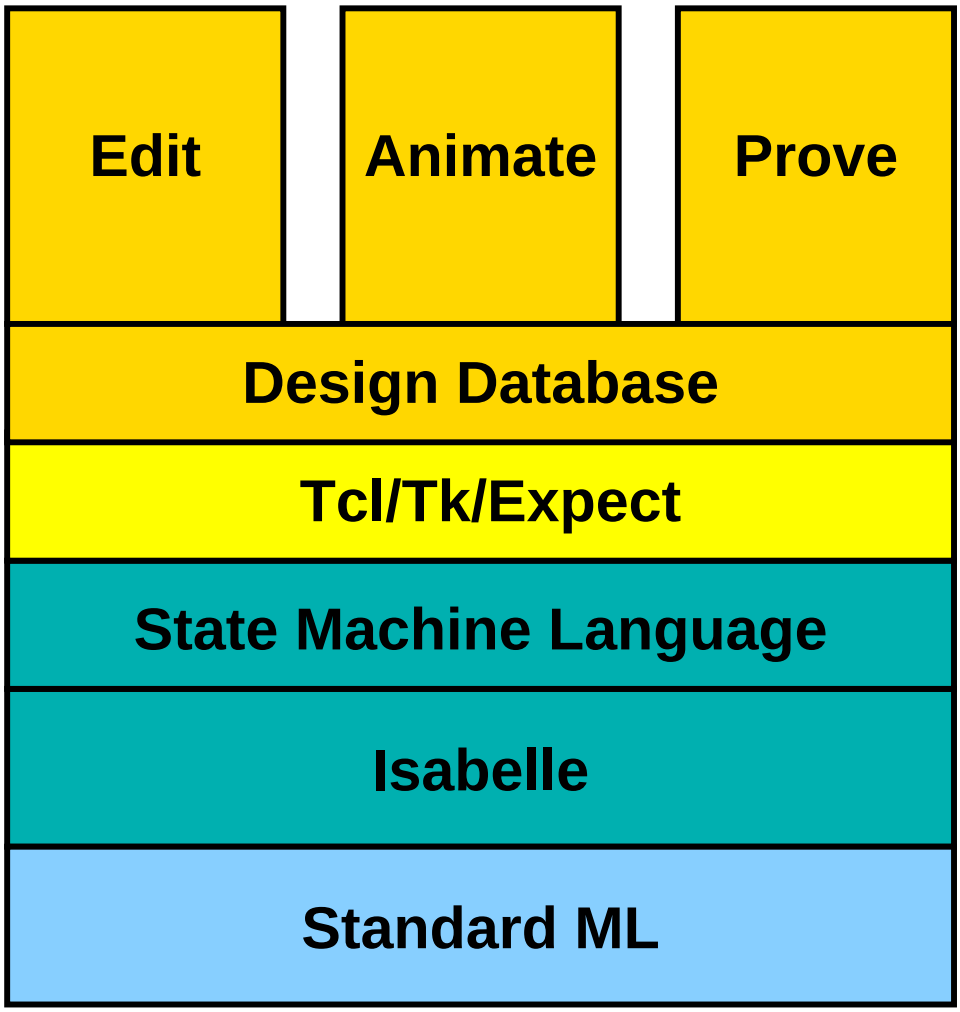


Figure 1.1: DOVE processes block diagram

- Each state machine has a number of attributes or variables (see Section 2.1) used to record information for future computations and to communicate with the environment. For example, in the traffic-lights state machine, the variable `NLight` records the colour currently desplayed by the north facing traffic light.
- In DOVE the control logic of a system is modelled by its state/transition graph. The graph consists of a number of state nodes, represented as circles, and a number of transitions between states, represented as line vectors.
- Each state node represents a decision point in the behaviour of the state machine. The transitions out of each node represent the possible next actions in each state. For example, in the `AllRed` state the traffic-lights state machine may either perform the `EWChangeGreen` transition and go to the state `EWGreen` or else perform the `NSChangeGreen` transition and go to the state `NSGreen`.

The traffic lights example will be used extensively throughout this manual and appears in full in Appendix A.

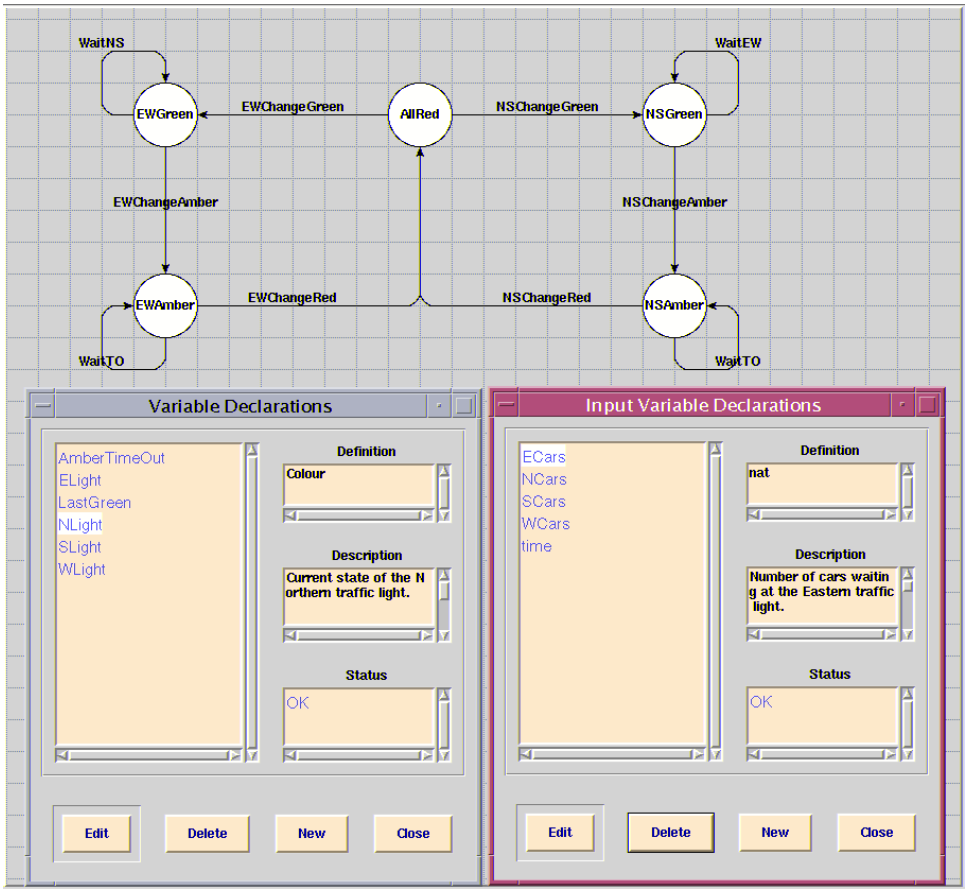


Figure 1.2: DOVE state machine structure

The DOVE system provides three basic components for treating state machine designs.

1.4.1 The editor

The state machine editor provides an intelligent, graphical environment dedicated to the construction of state machine designs. The transition graph of a state machine is constructed by laying out nodes and edges on a grid using simple point-and-click operations. Elements of the graph may also be moved around, modified, or deleted using conventional mouse-based commands. The editor uses intelligent graph-layout algorithms to provide a visually pleasing and readily comprehensible rendering of the transition graph. The logical relationships between transition edges and state nodes are maintained automatically during modification.

The graphical rendering of the state machine produced by the editor provides a vital aid to the comprehension and analysis of the design. An important feature is the editor's ability to generate high-resolution renderings suitable for presentation to a wide range of stakeholders. The on-screen rendering is also used extensively to inform the design-analysis activities of the other DOVE components.

1.4.2 The animator

DOVE provides symbolic animation features that can give increased confidence in a design. A DOVE animation is a 'what-if' style experiment on the state machine, which investigates the way in which the state machine evolves in a given situation. Each animation experiment consists of performing a sequence of transitions from some initial configuration of the state machine. Animation thus serves as a form of rigorous design-validation.

The animation process is performed by manipulating the transition graph rendered by the editor. An initial situation is installed by interacting with the graph and the desired execution path is determined by mouse-based point-and-click operations. The animator determines the result of the chosen experiment, interactively updating the display to reflect the results of each execution step.

1.4.3 The prover

The most powerful aspect of the DOVE tool is its ability to formally verify the properties of state machine designs. The aim of verification is to demonstrate logically that a state machine design satisfies a given collection of requirements. Each requirement is expressed in a formal language, specifically designed to support the description of the behavioural properties of state machines. The prover component provides a powerful environment for proposing, proving, and maintaining the consistency of these properties.

An important aspect of the DOVE proof process is the visual feedback that is offered. The transition graph is used extensively as a way of informing the user about the current proof state. The proof itself is also presented in a graphical mode which supports powerful high-level navigation and manipulation features.

Chapter 2

DOVE state machines

The state machine is a model of computation. The control flow of the model is determined by a graph in which the nodes represent decision points and the linking transitions represent the corresponding allowed choices. Experience has shown that state machine diagrams are an effective means of presenting system designs to a wide range of audiences. Tracing possible control paths by following transitions from an initial state to a final state is a natural and intuitive way of comprehending a design. If states and transitions are given properly evocative names, the reader can quickly gain a strong intuition about the purpose and intended behaviour of a state machine design.

For example, consider the state machine diagram for the traffic-lights example represented in Figure 2.1 (see Appendix A for more details). It is relatively straightforward to determine the basic behaviour of the system just from looking at the diagram. Computation begins by performing some initialisation action, then progresses to the `AllRed` state. From here, either the E/W or the N/S lights can cycle through green to amber to red, at which point the system is back in the `AllRed` state.

Unfortunately a state machine diagram is a long way from being a machine-executable specification of system behaviour. Moreover, even 'properly evocative' names can be misleading to some readers. For example, in the traffic-light diagram, there is no way of determining from the graph and description above exactly what happens during initialisation – or even of being absolutely sure that the N/S lights are not affected by the `EWChangeGreen` transition. In order to address these problems, DOVE augments the state machine mechanism with programming features that have well-defined, formal semantics. In this way, the state machine diagram can be made to serve as an invaluable aid to comprehension within a high-assurance design process.

This chapter presents a brief introduction to the way in which DOVE augments the concept of state machine diagrams and the way in which this is used to provide assurance of critical properties of state machines. Although the material in this chapter is presented in an informal way, it should be remembered that in the DOVE tool these concepts are formalised using the Higher Order Logic (HOL object logic) of the Isabelle proof tool [9]. This occasionally has impacts on the features available in DOVE, which will be emphasised in footnotes for the benefit of the informed reader.

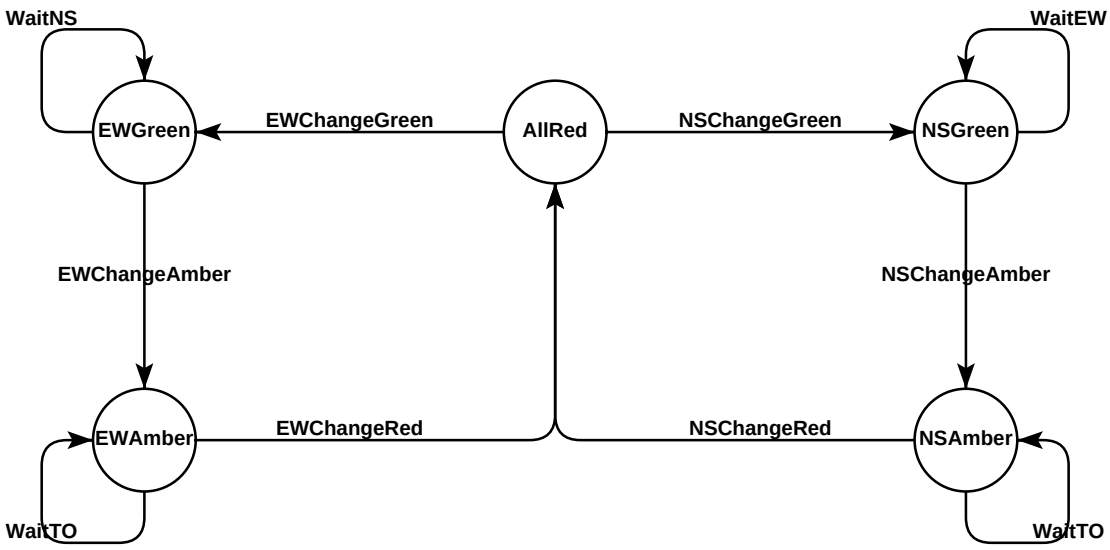


Figure 2.1: state machine graph for the `TrafficLights` *example.*

2.1 System attributes

In order to increase the fidelity of state machine models, DOVE introduces the notion of *system attributes*. Attributes are observable quantities associated with the behaviour of the system. In a DOVE state machine, these attributes fall into three distinct categories.

- *Input variables* are introduced by the user to represent the points at which external entities (the external environment) may influence the machine evolution.
- *Heap variables* are introduced by the user to store outputs to the environment or intermediate results in the executions of the machine.
- The current state and the last transition, the observable aspects of the control logic at any point in an execution of the machine, are tracked automatically by the DOVE tool via the state machine graph input by the user.

The heap and input variables are referred to collectively as the *memory*. The collected values of all the system attributes at a particular point in a computation is called a *configuration* of the system.

The input and heap variables used to help describe the traffic-lights example are presented in Table 2.1. The heap includes such observables as the current colour of each of the lights (`ELight`, `WLight`, `NLight`, and `SLight`), the last direction to show green (`LastGreen`), and the time at which an amber light is to change to red (`AmberTmOut`). The input variables record the number of cars waiting at each red light (`ECars`, `WCars`, `NCars`, `SCars`) and the current time (`time`).

An important point to note about Table 2.1 is the fact that the DOVE state machine language is strongly typed. Each of the system attributes is assigned a type which determines the kind of information it represents. The type `nat` is a standard Isabelle/HOL type and represents the natural numbers. The other types in the traffic-lights example (`Direction` and `Colour`) are enumerated

Variables	
Name	Type
LastGreen	Direction
ELight	Colour
WLight	Colour
NLight	Colour
SLight	Colour
AmberTmOut	nat

Inputs	
Name	Type
ECars	nat
WCars	nat
NCars	nat
SCars	nat
time	nat

Table 2.1: System attributes for the traffic-lights example.

types (Isabelle/HOL datatypes) which have been defined locally to the state machine. The definition of local types is discussed in more detail in Chapter 4. In this example, Colour has values Red, Amber or Green, while Direction can be NS or EW.

2.2 Transitions

The power of system attributes in DOVE state machines is that they allow each transition to be associated with a simple program. Each such transition program consists of three parts.

The Let: part allows the definition of abbreviations which can be used to simplify the transition definition. This is explained in more detail in Chapter 4.

The Guard: part, also referred to as the *guard*, of a transition describes a Boolean condition on the system attributes (both heap and input variables) which must be satisfied for the transition to occur. A state machine can only perform (*fire*) a transition if it is in a state from which the transition is allowed *and* the system attributes satisfy the guard.

The Act: part, also referred to as the *action*, of a transition describes the way in which the system's memory is changed when the transition fires. The action is expressed as a parallel assignment to a number of heap variables. Only the variables assigned to by the action may change. Input variables are updated independently by the environment, and so can never be assigned to in any action.

All sections of the transition's definition are optional: if the **Guard** is omitted, it is defined as **True** (i.e., the transition always fires), and if the **Act** part is missing, it will be defined as **Skip** (the transition does nothing).

As an example, the definition of the EWChangeGreen transition[1] is as follows.

```
Guard: (LastGreen = NS)
Act: ELight ← Green
     WLight ← Green
```

[1] Note that DOVE is fairly lenient in its handling of the syntax for transitions. In particular, reserved words such as **True**, **False**, **If**, **Then** and **Else** are case-insensitive, while the identifiers **And** and **Or** (also case-insensitive) may be used in place of the symbols & and |, respectively. Whitespace is ignored.

The transition guard is that the LastGreen variable should be NS; i.e., that the N/S lights were the last to be set green. The effect of the transition is to set the E/W lights to green.

The "wait-busy" transitions, such as the WaitTO transition, are required to prevent "deadlock". This is elaborated in Section 2.9 below.

2.3 Initialisation

Each DOVE state machine must have at least one state. Exactly one state in the state machine must be defined as the initial state (in terms of execution); this should be defined in the Initialisation Window, invoked via the Initialisation option below the Definitions menu item. This window must also be used to specify the initialisation condition, which acts as a requirement on the initial values of the heap variables. The initialisation need not fully determine the value of every attribute, since the initial values of some attributes may not be important to the correct behaviour of the state machine. If no particular initialisation is required, the initialisation condition may be set to True.

The initialisation condition for the traffic-lights example is as follows.

(NLight = Red) And (SLight = Red) And (ELight = Red) And (WLight = Red)

It requires that all the traffic lights initially show red. This means, for example, that the heap variable LastGreen may initially be either NS or EW. It does not matter which set of lights goes green first, provided they alternate correctly thereafter.

2.4 State machine definitions

A state machine design is completely defined by:

- its state machine diagram,
- its initial state,
- its system attributes,
- its initialisation predicate, and
- its transition definitions.

The minimum requirements for a legal state machine definition in DOVE are as follows:

- There must be at least one node.
- Exactly one node in the state machine must be defined as the initial state of all machine executions.
- There must be an initialisation condition, which is a Boolean expression.

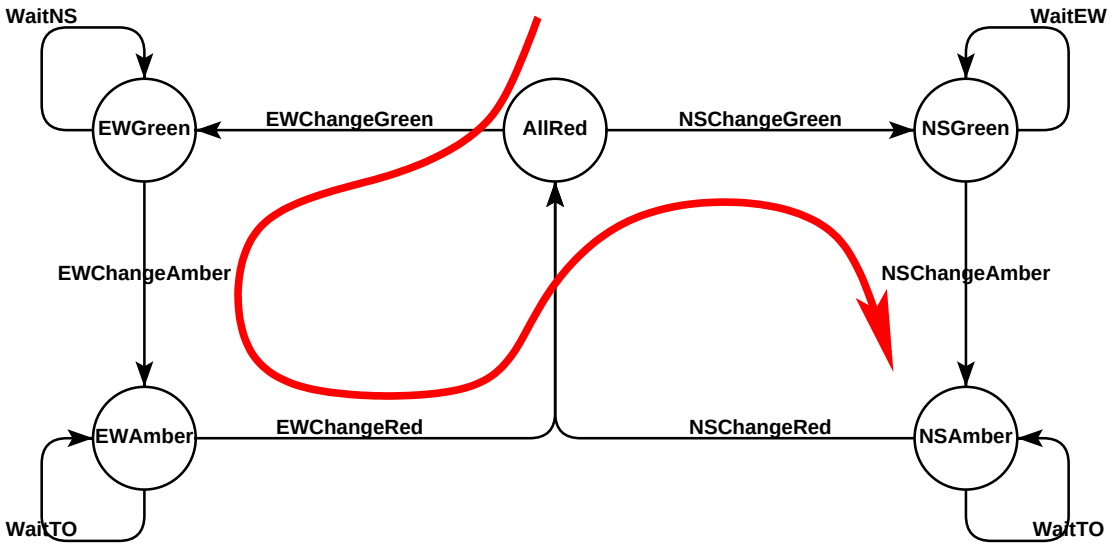


Figure 2.2: A possible execution path for the `TrafficLights` *machine*

state	AllR	→	EWG	→	EWA	→	AllR	→	NSG	→	NSA
LastGreen	NS		NS		NS		EW		EW		EW
NLight	Red		Red		Red		Red		Green		Amber
ELight	Red		Green		Amber		Red		Red		Red

Table 2.2: A possible execution of the `TrafficLights` *machine*

2.5 Executions

A DOVE state machine is interpreted as operating in the following manner.

An initial version of the memory is constructed so as to satisfy the initialisation condition and the machine is placed in chosen initial state. At each subsequent point in the computation the current state and memory are consulted to determine which transitions are enabled. A transition is enabled if and only if the state machine graph allows the transition from the current state and the current memory satisfies the transition's guard condition. If more than one transition is enabled, either may fire. In this way, the machine proceeds to trace a path through the state machine graph, as depicted in Figure 2.2.

As execution moves around the state machine graph, a sequence of state machine configurations is generated, one for each state visited. Such a sequence of configurations is called a *history*. A history which can be constructed by a given state machine is called an *execution* of that state machine. The execution associated with the execution path described in Figure 2.2 is depicted in Table 2.2.

An important property of state machine executions is that they are *prefix closed*; i.e., that if a given history is an execution, then so are all of its prefixes. In other words, the histories obtained by successively dropping the final configuration from an execution are all executions. This is because each new state machine execution is generated by extending some existing execution with the next allowed transition and state.

2.6 Animation

As described above, one simple way to comprehend a state machine diagram is to trace various execution paths visually. Such an inspection actually corresponds to the generation of a possible (partial) history of the state machine. The history traced need not be complete because it is not necessary to commence in the initial state. The history may not represent an actual execution of the state machine, because it is possible to trace paths on the diagram which are logically disallowed by the transition guards. For example, the `TrafficLights` machine cannot perform `EWChangeRed` and then `EWChangeGreen` because the value of `LastGreen` will disallow this. With the addition of system attributes and transition guards, it is possible to use this informal mechanism of path tracing to generate a great deal of useful information about a state machine design.

The enhanced version of path tracing supported by DOVE is called *animation*.[2] The technique consists of choosing a beginning state and tracing a sequence of contiguous transitions from that state. DOVE supports this technique by calculating the new values of any heap variables that are updated and (crucially) by calculating what is termed the *path condition*. The path condition is a Boolean expression which represents the logical requirements on beginning configuration for the traced path to be possible (only *possible* because other paths may also be possible if more than one transition guard is enabled). If the newly calculated path is not possible (i.e., if the guard governing the most recently fired transition actually prohibited the firing of that transition in the current configuration of the state machine), the path condition will be *false*.

The animation mechanism is a useful way of ensuring that all variables are updated as expected in critical situations, and for confirming intuition about when certain execution paths are possible or impossible. In this way, assurance of the correctness of the state machine design can be promoted to a high level in an intuitive and efficient manner. However, since animation corresponds essentially to a sophisticated form of testing, animation cannot provide the highest levels of assurance of state machine correctness. The highest levels of assurance can only be gained by mathematical proof that the design satisfies its requirements.

2.7 Properties

In DOVE, high-level system requirements are represented as collections of mathematical properties on state machine histories. Each property generates the collection of histories which *satisfy* the property. A state machine is said to satisfy a property if and only if every execution of the state machine satisfies the property. In the following we introduce the language used to describe system properties and illustrate the evaluation of such properties on the `TrafficLights` execution depicted in Table 2.3. We refer to this table as the *truth table* for the properties on this execution.

In order to specify properties easily, DOVE adopts a form of *temporal logic*. The aim of temporal logics is to provide a convenient notation for describing properties of *entire* histories. The DOVE temporal logic provides facilities for inspecting the values of all variables in the current configuration, in any previous configuration, and, importantly, for constraining the order in which events can occur.

[2] Note that in general the term animation may refer to a much wider range of techniques than that supported by DOVE.

state	AllR	→	EWG	→	EWA	→	AllR	→	NSG	→	NSA
LastGreen NLight ELight	NS Red Red		NS Red Green		NS Red Amber		EW Red Red		EW Green Red		EW Amber Red
NLight = Green	F		F		F		F		T		F
At NSGreen Implies NLight = Green	T		T		T		T		T		T
Previously (NLight = Green)	T		F		F		F		F		T
First	T		F		F		F		F		F
Initially (NLight = Red)	T		T		T		T		T		T
Always (NLight = Red)	T		T		T		T		F		F
Sometime (NLight = Green)	F		F		F		F		T		T
MostRecently (At AllRed) (LastGreen = NS)	T		T		T		F		F		F

Table 2.3: Some properties evaluated on an execution of the `TrafficLights` *machine*

The simplest properties are written as Boolean expressions on the current values of the machine attributes. For example, the expression

```
NLight = Green
```

can be used to determine if the north light is green in the current configuration. Table2.3 shows the result of evaluating this expression in each of the configurations of the execution path shown in Figure 2.2. The only state in which this is true is `NSGreen`. From such *configuration properties* we can build up a whole first order predicate logic using the usual operators `Implies`, `Not`, `ForAll`, etc.

So as to allow access to the current state and the last transition, the special temporal logic constructors `At` and `By` respectively are introduced. For example, the expression

```
(At NSGreen) Implies NLight = Green
```

says that the north light must be green if the machine is currently in the `NSGreen` state. From Table 2.3 it can be seen that this property is true in every configuration of the example execution. This is because the property `NLight = Green` need only be true when the machine is actually in the `NSGreen` state (since otherwise the *assumption* to the implication, `At NSGreen`, is false, and thus the implication is true).

In the above examples, ‘currently’ means the last configuration of the history being considered. In order to gain access to earlier configurations, temporal logic adds special operators called *modalities*.

The Previously modal operator asks if a property was true of the history gained by removing the last configuration from the current history. For example,

```
Previously (NLight = Green)
```

says that the north light was green in the previous configuration. From Table 2.3 it can be seen that the truth table for this property is just that of NLight = Green shifted along one column, with the addition that the property is true in the first configuration.

The constant First can be used to determine if the current configuration is in fact the first one. The truth table for First is also shown in Table 2.3. The value of all Previously properties is taken to be true in the first configuration, by convention. Since this convention is essentially arbitrary, it is a good idea to make all Previously properties conditional on not being in the first configuration. For example, it is better to write

```
(Not First) Implies Previously (NLight = Green)
```

since this makes it explicit that the Previously property is not very meaningful in the first configuration.

The Initially modal operator determines if something is true in the first configuration. For example,

```
Initially (NLight = Red)
```

says that the north light is red in the initial configuration. Referring once again to Table 2.3, it can be seen that this property is true in every configuration because it is true in the first.

The Always and Sometime modal operators allow the consideration of all the configurations in a history. For example,

```
Always (NLight = Red)
```

says that the north light must be red in every configuration of the history and

```
Sometime (NLight = Green)
```

says that it must be green in at least one configuration of the history. Once again the truth tables for these properties are depicted in Table 2.3. It is important to note from these tables that once an “always” property becomes false it remains false in all future configurations, and once a “sometimes” property becomes true it remains true.

The MostRecently operator is important in selecting points of particular interest in a history. It takes two arguments which are properties: the first selects the points of interest and the second describes a desired property. For example,

```
MostRecently (At AllRed) (LastGreen = NS)
```

says that the last time that the machine was in the AllRed state, the LastGreen variable had the value NS. The truth table for this property shows that it is true for the first three configurations and then false for the last three, because the value of LastGreen is EW when AllRed is visited the second time.

There are several other modal operators supported by DOVE, but the above are the most commonly used. A full account of DOVE's temporal logic is presented in Appendix C.

2.8 Verification

Verification in DOVE corresponds to proving that all executions of a state machine satisfy a required property. This condition is represented using the *turnstile* operator. For example,

```
|- Always (At AllRed Implies ELight = Red)
```

claims that the property `At AllRed Implies ELight = Red` is actually true of the traffic-lights machine. For technical reasons that are discussed in Chapter 7 it is important to apply the `Always` operator to all desired machine properties in DOVE.

The basic proof technique in DOVE (the mechanics are discussed in detail in Chapter 7) is *induction* on the executions of the state machine. Suppose that all the initial configurations satisfy the property. Suppose, further, that for every execution of the state machine which satisfies the property, all enabled transitions take the state machine to an execution which also satisfies the property. Then the property must hold for all executions.

What is particularly powerful about DOVE is the fact that the induction step is reinterpreted as a simple statement in temporal logic. For any given temporal property, DOVE is able to calculate a property which describes what must have been true in the previous state in order for the current history to satisfy the given property. This is done by a process called *back-substitution*.

The basic idea in back-substitution is to replace occurrences of attribute names with the values assigned to them by the last transition. For example, if the last transition was EWChangeRed, then the property

```
At AllRed Implies ELight = Red
```

is back-substituted to the (trivially true) property

```
AllRed = AllRed Implies Red = Red
```

because the edge with transition EWChangeRed ends on AllRed, after which ELight is set to Red. The details of how the last transition information is determined are discussed in Chapter 7.

In general, back-substitution through a given edge E, applied to a given property ϕ, returns the weakest property which ensures ϕ; i.e., the largest set of histories for which the execution extended as determined by E satisfies ϕ. The result is known as the *weakest temporal precondition*. It is obvious that, as discussed above, this is obtained for a configuration property by replacing variables with the values determined by the transition in edge E. Since a general property is generated from

configuration properties by the modal operators, to determine its weakest temporal precondition we simply need to know how to distribute back-substitution through the various modalities. Here we just list a few[3] of the results, writing backsub to represent the operation of back-substitution through a given edge:

- backsub T = T ;
- backsub (ϕ And ψ) = (backsub ϕ) And (backsub ψ) ;
- backsub (Not ϕ) = (Not (backsub ϕ)) ;
- backsub (ForAll x • ϕ x) = ForAll x • (backsub ϕ x)
- backsub (Previously ϕ) = ϕ ; and
- backsub (Always ϕ) = ((backsub ϕ) And (Always ϕ)) .

As a simple example, consider a property of the form

Previously Always ϕ .

The action of back-substitution leaves

Always ϕ .

At this point it is convenient to extract the current state information by expanding via the simple rewrite

(Always ϕ) = (ϕ And (Previously Always ϕ)) .

To be specific, the property

Previously Always (At AllRed Implies ELight = Red)

is back-substituted to

(At AllRed Implies ELight = Red) And
Previously Always (At AllRed Implies ELight = Red) .

This kind of result will be used repeatedly in Chapter 7.

Once the back-substitution of a required property has been determined, the induction step becomes a ‘simple’ matter of proving that the required property implies its own back-substitution – remember that the back-substitution of a property ensures that property will be true in the next state. The process may be a bit more complicated than this in practice, because it may be necessary

[3]The knowledgeable reader will perhaps be surprised at the simple distribution through Not. However, the back-substitution is only applied for the action part of the transition, which – by construction as a list of variable assignments – is deterministic. This provides great simplification in the distribution properties and is crucial to the efficiency of verification in the DOVE tool.

to back-substitute several times before the implication can be proved. Such a computation is equivalent to generalising the induction hypothesis to extension by a history of several steps instead of the usual one. For uniformity of the tactic presentation we have extended the notion of back-substitution to also apply to the initial case of the induction proof, where its action is simply to insert the state machine's initial predicate into the hypotheses and attempt to show the resulting initial configuration property. For this reason it is necessary to decompose the property under `First` or `Not First` before the edge decomposition (in the `Not First` part) required for the back-substitution described above. Several examples of such inductive proofs will be found in Chapter 7.

2.9 Scope of DOVE

From the previous discussion it is clear that in DOVE we restrict our attention to properties that can be represented as a set of finite sequences of configurations. This restriction is crucial to the effectiveness of this back-substitution technique. It also has important implications for the kinds of properties that can be treated by DOVE. The following is somewhat technical, and the impatient reader may wish to omit this section on a first reading.

The first point to make is that this choice implies a restriction to only those properties which can be determined by consideration of individual execution traces. Many important system properties can only be determined by consideration of *all* the possible executions. A simple example of such a property is *determinism*. Determinism requires that there is exactly one allowed response to any given sequence of inputs. Clearly this cannot be established simply by considering individual executions. (Actually, by first reasoning abstractly about the structure of state machines it is possible to show that for any given state machine there is a simple property which corresponds to determinism, namely that only one guard is enabled at each point in every execution.) Another important example of a whole-system property is the non-interference privacy-property of security critical systems. Such properties cannot be treated directly by DOVE-style verification, and can only be treated indirectly when an equivalent simple property can be determined by abstract argument (as in the case of determinism).

The literature identifies two major categories of simple properties [4]. A property is a *safety* property if and only if it can be falsified in a finite time. This means that a safety property can be described fully by the finite histories which satisfy it. An infinite history satisfies a safety property if and only if all of its finite prefixes satisfy the property. That is to say that safety properties are prefix closed. Alternatively, a property is a *liveness* property if and only if it can *not* be falsified in a finite time. This means that a liveness property cannot be characterised properly by the finite histories which satisfy it – i.e., liveness properties are not prefix closed.

Since DOVE deals solely with finite histories, the class of properties treated by DOVE is exactly the safety properties. This choice has been made in order to make property verification in DOVE as simple as possible. However, as now discussed, provided the right timing model is adopted, it is possible to do without liveness properties.

To see this, consider the purpose of liveness properties. Liveness properties are often characterised informally as those properties which require a computation to make 'good' progress, and safety properties as those properties which ensure no 'bad' progress is made. If these intu-

itive characterisations were correct absolutely, then an inability to treat liveness would be a major weakness for DOVE.

However, it is simply not true that ‘safety’ in critical systems is purely about ensuring no ‘bad’ progress is made. Sometimes, as when responding to a dangerous situation, it is critical that ‘good’ progress be made, while making ‘no progress’ is ‘bad’ progress. This is one of those cases where the use of natural language can lead to considerable confusion due to different meanings of a word (in this case ‘progress’) being used close together and due to unstated assumptions. The unstated assumptions in the above characterisations of safety and liveness are that a failure to progress is possible and that it can never be ‘bad’; i.e., ‘no progress is good progress’. In practice, some form of progress is always made, because of the ‘endless march of time.’

The question of whether ‘no progress is good progress’ can be side-stepped by adopting a clockwork model of time, that is by identifying the progress of the state machine with the progress of (abstract) time. This ensures that progress is always made, because time is always passing. Under this assumption, safety properties can be used to approximate any desired liveness property[4]. For example, instead of requiring that a green light *eventually* turn red, require that it turn red within a certain number of transitions or, more generally, before some increasing attribute (such as clock time) reaches a certain value.

Thus, the minimum requirement for treating liveness properties through safety approximations is that the state machine is always able to perform some transition – such a machine is said to be *deadlock-free*. A deadlock-free state machine always makes some progress, effectively disallowing the ‘no progress is good progress’ option. Any given state machine can always be converted to deadlock-free state machine by adding ‘busy-wait’ transitions which fire when no other transition is enabled. For example, this is the purpose of the `WaitTO` transition in the traffic-lights example.

It is important to note that DOVE itself does not enforce a clockwork model of time. The designer must explicitly model any aspects of time which are critical to statement and/or proof of critical properties of the state machine. The more sophisticated the progress property to be proved the more sophisticated must be the model of time imposed on the state machine. The DOVE tool offers no help to the user either in determining the required model of time or in imposing it on the state machine; but it does allow the user the freedom adopt the model best suited to their particular problem.

[4]For the technically minded, this fact follows from the status of liveness properties as limits of sequences of safety properties, much as the irrational numbers are limits of sequences of rational numbers.

Chapter 3

A first look at DOVE

This chapter provides a brief introduction to the DOVE tool. Detailed discussions of the components introduced here appear in later chapters. Before starting, DOVE and the other required packages must have been installed according to the instructions in the file `Install.ps` found in the distribution documentation. It is explained in those instructions how the user must add DOVE's bin directory to the user's path variable and how to overide the set of standard resources (eg colours, fonts, window geometries and so forth) chosen by DOVE. If required, further detailed information on acquiring and installing DOVE can be found at the DOVE web site (`http://www.dsto.defence.gov.au/esrl/itd/dove`).

3.1 Starting DOVE

Figure 3.1 shows the DOVE state machine window of the example TrafficLights state machine on startup, and the following should be read with this figure in mind.

The syntax for starting DOVE can be summarised as:

```
> dove [-- -h | [machine]]
```

The following options are supported.

- `-h`, prints out this list of options.
- `machine`, where "machine" stands for the user-chosen machine name.

The example in Figure 3.1 was started by the command

```
> dove TrafficLights
```

The beginning user may like to experiment with this state machine package, which is included with the DOVE installation in the directory `Examples/TrafficLights`. Typing

```
> dove TrafficLights
```

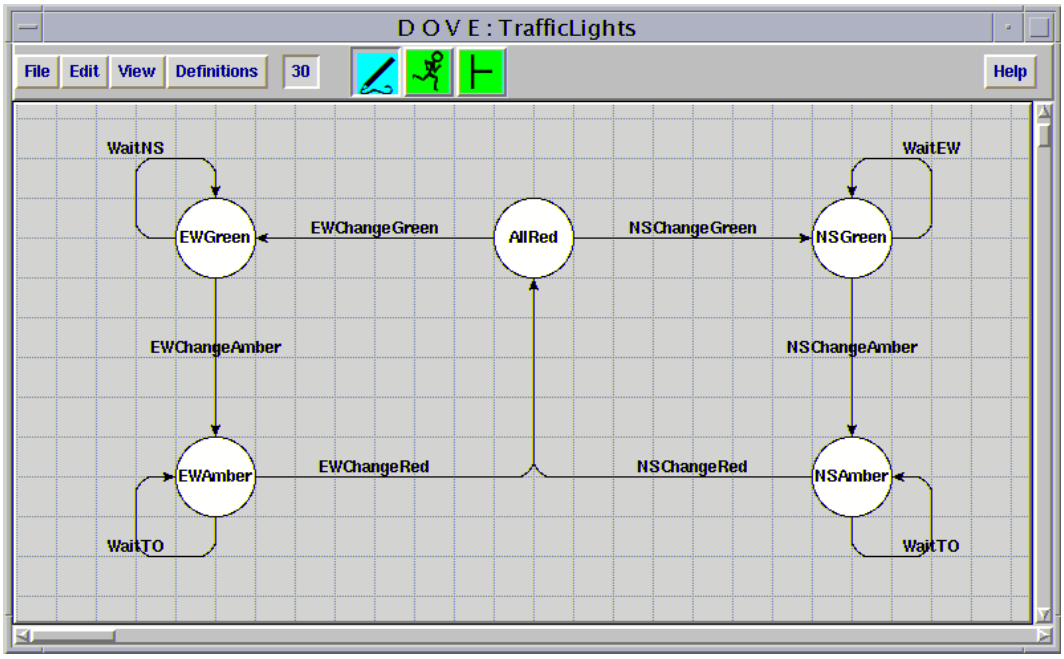


Figure 3.1: DOVE state machine window at startup.

in this directory will open the state machine graph with default settings. Alternatively, the user may simply type

```
> dove
```

and a DOVE session will start with a blank grid, ready for defining a new state machine.

3.2 DOVE files

DOVE makes use of a number of local files, which are generated in the course of a DOVE session. This section briefly describes what the files are and how they are used, but first the standard DOVE file-structure conventions are considered. The impatient user who is not interested in the subtleties of the various files should just read the preamble about file management, and then skip to Section 3.3 before proceeding to the later chapters.

The first step is to decide on a working title for the system design under consideration, and then create a so-named working directory to hold the files for this state machine. The DOVE tool should then be executed from the working directory, and all the associated DOVE files should be stored there.

In the case of the traffic-lights state machine example, the working title of the design is `TrafficLights`. In order to build this design, a directory `TrafficLights` is created.

```
> mkdir TrafficLights
```

The DOVE tool is then invoked from this directory.

```
> cd TrafficLights
> dove TrafficLights
```

By convention, all the DOVE files associated with the TrafficLights design have names beginning with TrafficLights. For example, a typical listing of the TrafficLights directory after a DOVE session would be as follows.

```
> /bin/ls
TrafficLights.ML          TrafficLights.polyml-3.X  TrafficLights_graph.ps
TrafficLights.nw          TrafficLights.smg
TrafficLights.pdf         TrafficLights.thy
```

In the remainder of this section the purposes of these files are considered in detail.

3.2.1 State machine graph file

All of the information required to describe the state machine is stored in a file with an smg suffix, standing for 'state machine graph' (for example, TrafficLights.smg), which includes the information needed to both draw the state machine graph and to determine which transitions join which states. Furthermore, it includes all of the mathematical definitions necessary for generating the logical theories which are used to reason about the state machine.

It is generally inadvisable for the user to edit the smg file by hand.

3.2.2 Noweb files

The noweb package is a literate programming environment which DOVE uses to produce the documentation files discussed in the following subsection. The file TrafficLights.nw, written by DOVE during the compilation process, contains the instructions noweb needs to create the documentation files and to insert the indexing and cross-referencing information.

3.2.3 Theory files

The logical theory associated with the state machine is stored in two files. The first has a name of the form *.thy (for example, TrafficLights.thy) and contains an Isabelle theory definition describing the state machine. The second has a name of the form *.ML (for example, TrafficLights.ML) and contains commands, in the ML programming language, which provide a customised Isabelle environment for reasoning about the state machine. Both of these files are generated from the noweb file.

3.2.4 Image file

In order to make efficient use of the state machine theory, the Isabelle commands in the *.thy and *.ML files are compiled into an ML image file which is given the base-name of the state

machine and the extension name of the ML compiler for which it was created (for example, `TrafficLights.polyml-3.X`). The image file is an executable file, and is invoked whenever DOVE begins to prove a property.

3.2.5 High-resolution documentation file

The DOVE tool is able to output high-resolution documentation in Portable Display Format (PDF). The documentation is placed in a file of the form `*.pdf`. The documentation file includes extension hyper-indexing and cross-referencing of the various definitions in the state machine as well as recording any remaining parsing errors. This makes it an invaluable tool in understanding and debugging a design.

3.3 DOVE tools

When started as discussed above, the DOVE session begins in the DOVE state machine window exemplified by Figure 3.1. The grid section, called the “canvas”, is used for editing graphs and the presentation of animation and proof visualisations.

DOVE has three modes of operation: edit, animation, and proof. The three mode buttons in the menu bar of the DOVE state machine window are used to move between these modes. If the default colour scheme is being used then the buttons are either blue or green, blue indicating the current mode. The initial setting for the DOVE session is edit mode. As a further visual aid, the mouse cursor appears differently in the different modes, as listed below.

(a) Edit

(b) Animate

(c) Proof

Figure 3.2: DOVE mode buttons

3.3.1 Edit mode

The design of state machines is carried out in the edit mode, which can be entered by clicking on the edit (“pencil”) icon (Figure 3.2(a)) on the menu bar. The operations that can be performed in edit mode are discussed in Chapter 4.

In edit mode the cursor appears as the default mouse-cursor.

3.3.2 Animation mode

Animation mode is used to observe how variables evolve symbolically during execution of the state machine. It is entered by selecting the button with the picture of a runner (Figure 3.2(b)). The operations that can be performed in the animation mode are discussed in Chapter 5.

In animation mode the cursor appears as an arrow pointing right.

3.3.3 Proof mode

The proof mode has two main functions: to enable the definition, editing and inspection of proof queries; and to carry out a proof of a particular query. It is entered by selecting the button with the picture of the turnstile (Figure 3.2(c)). The operations that can be performed in the proof mode are discussed in Chapter 6.

In proof mode the cursor appears as a turnstile.

Chapter 4

Editing the state machine

The DOVE state machine window is the main interface to the DOVE session. It appears upon startup, with the name of the currently displayed state machine graph shown in the title bar. The structure of the window is clear from the example in Figure 3.1, and the user should refer to this figure when reading the following sections.

The specification of a state machine in DOVE requires

- the graph of the state machine, consisting of nodes and edges, to be drawn in the DOVE state machine window, and
- the variables and transitions which make up the state machine to be defined.

In this chapter these points are expanded in turn. The menu options will be explained after the discussion of graph drawing, since they are not needed in the geometrical construction. However, it is worth noting at this point that there is a menu option for undoing and redoing changes in the graph, as well as menu options modifying the display.

There are a number of rules that need to be followed when designing state machines in DOVE. These are discussed in Section 4.4 below. After a cursory examination of these, the "hands-on" reader may wish to move quickly to the tutorial session in Section 4.5, referring to earlier sections when required.

4.1 Graph editing on the canvas

The *canvas* refers to the grid-lined area under the menu bar. It is here that the graph editing operations are carried out, using the mouse and the keyboard. The canvas size is fixed, but the grid size can be changed in a given session (via the menu bar, as discussed below) and it is the grid size which fixes the scale size of the graph. Moreover, just a portion of the canvas is displayed in the window and scroll bars are provided to access the remainder. Thus, it is not expected that the available canvas area will impose any limitations on state machine design.

The basic objects of the state machine graphs are labelled *nodes*, and labelled, directed *edges* (circles and arcs, respectively). The nodes have four distinguished points, or *vertices*, at the cardinal points on the circle – i.e., at the N, S, E, W compass directions. All edge lines begin and end

at the vertices. The commands of the graph editing interface allow creation, deletion, movement, and renaming of the basic objects. A quick synopsis may be useful.

- Creation is done with mouse button 1, and deletion with Control-button-1 (i.e., holding the Control key down and pressing mouse button 1).
- Movement of objects is performed through clicking on the item with mouse button 2[5], moving to the desired location, and releasing mouse button 2.
- To rename an item, move the mouse-pointer onto the corresponding text. At this point a text cursor will start blinking – notice that pressing the mouse buttons is not required – and the keyboard can then be used for editing.

These commands are discussed systematically below.

4.1.1 Graph layout

The visible grid of the DOVE canvas is not just a guide to the user's eye, it is an integral part of the automatic geometric construction: all basic objects within a state machine graph are "snapped" to the grid.

- Once placed, a node automatically centers to the closest position on the grid, and adjusts its vertices to lie on the grid's axes.
- Edge lines lie on grid lines, as much as possible.

This leads to neat graph presentation without the user being required to perform fine-scale editing.

As a final preliminary, note that DOVE prevents the user from placing the basic graph objects too closely for neat graph layout. During any movement of an object the user is informed of the areas that the object cannot occupy. These areas, called *blocks*, are temporarily shaded in red during the move, as depicted in Figure 4.1. If the cursor enters a block during a move, the moving object is shown at its most recent legal position (on the edge of the block), and once the move is completed (by button release) the object will be created at the most recent legal position (snapping to the grid as normal). The shading is also displayed if the creation of an object violating the separation requirement is attempted. The shading is then removed once the illegal creation has been abandoned (i.e., once the button has been released).

4.1.2 Nodes

Nodes appear as solid white circles with a black outline, and with the node name as a text string centred on the circle. The centre of the circle is snapped to a grid point, and the node radius is equal to one grid spacing. The node then intersects the grid at four points, the node vertices.

[5] On a standard three-button mouse, mouse button 2 is the middle button. On a standard two button mouse, it is the right button. The tool is set up so that, in fact, mouse button 2 or mouse button 3 can be used interchangeably.

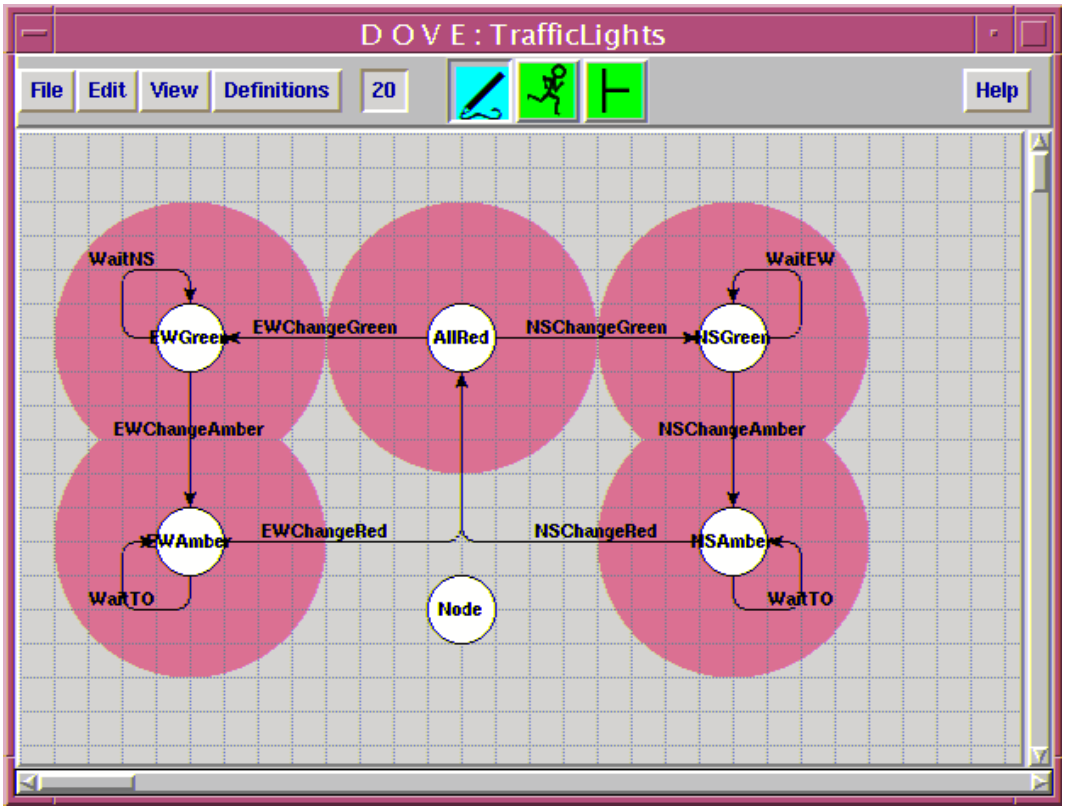


Figure 4.1: A DOVE state machine during node movement

node creation: To create a node, click mouse button 1 on an unoccupied area of the canvas. Recall that a certain minimum separation between nodes is required by DOVE. If this is violated, the areas which are too close to the point clicked on will be temporarily highlighted in red.

node deletion: To delete a node, hold the Control key down and click on the node with mouse button 1. Note that when the Control key is held down the mouse cursor becomes a "skull and cross-bones" to indicate that the system is ready to delete.

node moving: To move a node, place the mouse cursor inside the required node and press and hold mouse button 2. Move the node to the required location, and then release the button[6]. Whilst moving the node, those parts of the canvas that are too close to another node will be highlighted as red blocks and it will not be possible to drag the node into one of the highlighted areas (see Figure 4.1). When the button is released DOVE will redisplay all the edges attached to the node, and remove the red highlighting on the canvas.

node renaming: When first created, all nodes have the default name `Node`. The name can be changed by placing the cursor inside the node, at which point a blinking text cursor will appear at the end of the node's label allowing editing of the label to be carried out via the keyboard. When the mouse cursor is moved off the node, the new name will be registered. Nodes must have distinct node names.[7]

[6] From now on this collection of operations will be referred to as "*dragging* with mouse button 2".

[7] Precise rules for the allowed range of names are presented in Appendix B.

4.1.3 Edges

Edges are drawn as a connected sequence of black horizontal and vertical line segments whose ends are snapped to the grid. The junctions of the vertical and horizontal segments are drawn as a smooth arc. Edges are directed and labelled. The edge must begin and end at a node vertex; an arrowhead indicates the direction of the edge. The edge label appears as a text string in a position initially chosen by DOVE.

Different edges may have segments which overlap. To assist in distinguishing the different edges and corresponding labels, when the cursor moves onto an edge (or edge label) the entire edge and edge label are highlighted in blue. The highlighting is removed once the cursor moves off the edge or edge label.

Edge creation Holding down mouse button 1 while the mouse cursor is on a node, and then dragging with the mouse, will create an edge which leaves that node from the vertex closest to the point clicked on. If mouse button 1 is released while the mouse cursor is inside a node, the edge will be attached to the vertex closest to the release point. DOVE will then produce a snapped-to-grid path for the newly created edge. If mouse button 1 is released while the mouse cursor is over something other than a node, then the edge is not created and disappears. It is possible to create an edge leaving and entering the same node, but only at two different vertices on the node. Unusual kinks or wiggles in the newly created edge path can be straightened out by moving the corresponding edge label. It is good style to lay out the graph so that no kinks remain.

Edge deletion To delete an edge, while holding the Control key down click on the edge with mouse button 1. When the Control key is held down the mouse cursor becomes a skull and cross-bones to indicate that the system is ready to delete.

Edge moving DOVE considers edges as having three separate parts which can be moved independently: the start and finish points and the edge label. The user can experiment by clicking with mouse button 2 to see which region of the edge DOVE assigns to the different parts.

Moving edge labels An edge label can be dragged using mouse button 2, and will snap to the grid when released. Edge labels must not be too close to nodes, so the prohibited areas are displayed as red blocks during the move. Since the edges drawn by DOVE must go through the label, by moving the label the user has considerable control over the graph layout.

Moving edge endpoints Again using mouse button 2, an edge endpoint can be dragged to a new position. If mouse button 2 is released with the endpoint over a node, the new endpoint for the edge will be the vertex of that node closest to the release point. DOVE then produces a new path for the edge. The edge label can again be moved to achieve the desired layout. If mouse button 2 is released with the endpoint away from a node, then the edge reverts to its original path.

Edge renaming When first created, all edges have the default label[8] `Edge`. The name can be changed as for node renaming discussed above, the blinking text cursor being activated when the mouse cursor is placed anywhere on the desired edge or label.

4.2 The menu bar of the DOVE state machine window

The menu bar contains a number of pull-down menus and buttons. At a given point in the DOVE session a given menu option may not be available, as indicated by a "greying out" of its name in the menu. Options which communicate with the file directory are directed through a pop-up file-navigator window. Similarly, options which make declarations for defining the state machine are directed through a pop-up dialog box. Finally, certain command options require confirmation from the user. In the following subsections the options are discussed systematically.

4.2.1 The File menu

New clears the current state machine and makes ready for the design of a new machine. The user is asked to confirm the request if the current state machine graph is not saved. This option is only available during Edit mode.

Restore restores the currently loaded graph to the last saved version. The user is asked to confirm the request before the restore takes place. The option is not available if the current graph was started from a blank canvas and never saved, or if no modifications were made since the last load or save. This option is only available during Edit mode.

Load File presents the user with a file-navigator for loading the required state machine graph file. In doing so, DOVE clears the current state machine and loads the required state machine from the file. If the current state machine graph is not saved (i.e., it has been modified since the last save or load) the user is first asked to confirm that the operation is to go ahead. The user may browse other directories to find other `*.smg` files by clicking in the `Directories` frame of the file navigator. The `Load` operation has the keyboard accelerator `Meta-L`. This option is only available during Edit mode.

Save saves the current state machine to the filename currently displayed on the DOVE state machine window title bar, without asking the user for confirmation. The option is available if the current graph has previously been saved to, or loaded from, a file, and if there has been some modification since the last save or load operation was performed. The `Save` operation has the keyboard accelerator `Meta-S`.

Save As presents the user with a file-navigator for saving the current state machine to a specified file. This operation also renames the image file. This option is only available during Edit mode.

[8] Though called an edge label here, it is actually a label for the transition associated with the edge. This is relevant when the state machine is defined, see the menu item `Transition` in Section 4.2.4 for further details.

Create Documentation presents the user with a Create Documentation window which lists any known problems with the current state machine (if any), and provides two buttons:

- Cancel which cancels the documentation request,
- Create Documentation which continues.

If there have been modifications since the last save (or if the state machine has never been saved), the user is required to save the state machine before the Create Documentation window will appear. If the user chooses to continue after reviewing the Create Documentation window, then the documentation files discussed in 3.2 are produced, and DOVE starts up Acrobat – if it isn't already running – to display the PDF output.

Quit DOVE terminates the DOVE state machine window, and any other DOVE tools that the editor has started up (e.g., Animator or Prover). If there have been modifications since the last save or load, the user is required to confirm the action of quitting. The Quit operation has the keyboard accelerator Meta-Q.

4.2.2 The Edit menu

Undo undoes the last graph-editing operation (as discussed in Section 4.1) which altered the graph before its current display. This includes creating, moving and deleting nodes or edges. All of the modifications since the last save or load are kept in a stack, and can be undone in reverse order. On a Sun keyboard, the 'Undo' key may be used. This option is only available during Edit mode (animation has its own undo and redo mechanisms).

Redo re-performs the last operation which was undone. Until an operation is undone, this command is not available. All operations currently undone in the command stack can be redone, in reverse order. On a Sun keyboard, the 'Again' key may be used. This option is only available during Edit mode (animation has its own undo and redo mechanisms).

Check/Compile performs a variety of checks on the state machine, both structural and syntactic, largely to ensure that it has been designed in conformance with the rules in Section 4.4. Diagnostic messages are printed in a dialog box labelled Compilation - Diagnostics which comes up when this option is selected. A description of the possible diagnostic messages is given in Appendix D. It is required that the user applies the checks afforded by this option *before* proceeding to analyse the state machine. Indeed, the Prover will not come up, and the animation mode cannot be chosen, while fatal errors exist. Note that if no changes have been made since the compilation was successful then the compilation will *not* be repeated. Thus there is no further unnecessary time burden in the process.

4.2.3 The View menu

The View menu contains a number of operations that adjust the display.

Show Grid allows the canvas grid to be turned off, or redisplayed. Graph objects are still required to conform to the layout requirements of the grid even if it is not visible.

Zoom In increases the grid size by two.

Zoom Out decreases the grid size by two.

Set Zoom Level sets the grid size. It opens a small dialog box containing a sliding scale widget that the user can drag to select a new zoom level. Clicking the `OK` button dismisses the dialog box and changes the zoom level. Clicking `Cancel` dismisses the dialog and retains the previous zoom level.

4.2.4 The Definitions menu

An important part of the definition of a state machine is the specification of the corresponding components: the types, constants, heap variables, input variables and transitions which define the formal theory. In DOVE this is accomplished via the various options available in the `Definitions` menu. The majority of these options present the user with a dialog box in which the current declarations for that particular category (i.e. type, datatype, variable or whatever) are listed. For example, selecting the option `Transition Definitions` from the `Definitions` menu will invoke the `Transition Definitions` window, which both displays and enables modifications to the currently defined transitions for the state machine.

Selecting an item in the list of declaration names causes that item's definition and description to be displayed in other areas of the window. There are a number of ways in which list selection can be manipulated by the user. Clicking on a declaration name with the left mouse button is, of course, the most obvious way to make a selection. The user can also apply the `Up`/`Down` arrow keys in order to traverse the list, while the `Escape` key will clear the current selection. The list also features a case-insensitive search facility, which enables the user to type the first few characters of a declaration's name, at which point the first matching name in the list will be located and its declaration will be displayed. The user may then click to select it. To begin the search, which will proceed from any position in the list, first clear the current selection via the `Escape` key. When typing the name to search on, the `Backspace` and `Delete` keys will also function as expected. Please note that all of these key strokes will only be available while the mouse cursor is within the region of the list.

Each declaration window contains four buttons at its base: `Edit`, `Delete`, `New` and `Close`. The `Close` button simply withdraws the window; any declaration storage will already have been carried out. The `Edit` and `Delete` buttons are only available when an item has been selected. Neither `Edit`, `Delete` nor `New` are available when any declaration is in the process of being edited; that is, the user can only carry out one modification at a time. All declaration windows which modify the underlying formal theory of the state machine are disabled during animation and proving - this is clearly essential to maintain the integrity of the proof system. The declaration window brought up by `Property Definitions` is the `Properties Manager` window, in which the user stores properties defined *in* the formal state machine theory for analysis in the Proof

session. This is the only declaration window which does not modify the formal state machine theory, and indeed it *is* available in the Proof mode as well as the Edit mode.

The Edit button will invoke an editor window for the currently selected declaration. The editor window may differ slightly from one definition category to another, but in general it allows the user to specify:

- a new name for the declaration,
- the definition for the declaration, and
- a description/comment.

Editor windows also feature three buttons at the base: Close (which retains the declaration as the user has defined it, regardless of its validity), Commit (which attempts to parse the definition, and warns the user if the definition is invalid), and Cancel, (which discards any modifications that the user has made, and closes the window).

The declaration's status is affected by what happens during editing, and the value of the declaration's status is displayed in the area marked Status on the declaration window. The status can be one of the following:

- **unchecked:** The user has defined or modified the declaration, and has closed the editor window without attempting to parse/commit the declaration. By default, unchecked declarations are shown in black italics.
- **checked:** The user has successfully parsed/committed the declaration since the last modification. By default, checked declarations are shown in blue, plain font.
- **invalid:** The user has attempted to parse/commite the declaration, but the definition was found to be invalid. By default, invalid declarations are shown in red, bold font.

Note that it is only possible to compile the state machine (required prior to animation or proving) when all the declarations have been checked. The provision of unchecked or invalid declarations supports a more lenient user interface, in that a state machine can be saved for further work without having to be completely well-defined, and certain declarations (such as transitions) can easily be written without all their constituent parts having to be pre-defined. However, the user is strongly encouraged to commit most declarations, particularly those building blocks such as types and variables which may be referred to by other declarations, as they are declared. Otherwise, each declaration will need to be parsed separately prior to carrying out compilation. The ability to store unchecked or invalid definitions – simply by closing the editor window – has only been provided so that the user has the capacity to work in a more flexible manner; if abused, the user will ultimately find that too much flexibility may not pay off.

To define a new declaration, the user must activate the New button, which invokes an empty editor window, in which the user is required to enter the name, definition and description for the new declaration. Note that declaration names must be free of embedded white space. A further consideration is that – due to a bug in the Linux operating system – identifiers with a length of exactly 12 characters should be avoided by the user. This is a recommendation which should apply even when Linux is not being used, in the interests of maintaining the portability of state machine

designs. Once the new declaration has been defined, the user may either commit it immediately, or else store it for later use by closing the window, as discussed above.

The correct definition for a declaration will depend on its category. For example, constants and variables are defined simply by an appropriate type. The transition definition is more involved, as discussed in Section 4.3. The required syntax for declarations is that the basic "words" are strings of alphanumeric characters and underscores, which must begin with a letter. Names of type abbreviations, variables, constants, and all the elemental words in their declarations, are of this form. These rules are summarized at the beginning of Section 4.4. The full syntax for defining DOVE state machines is further explained in Appendix B.

Initialisation starts up a dialog box to enter the initial state for executions of the state machine, and the initial predicate which any initial configuration must satisfy.

Type Declarations starts up a dialog box to define abbreviations for the types of variables and constants. All Isabelle/HOL types and type constructors are available. Some simple examples include:

- `bool`: the Boolean type of truth values;
- `nat`: the natural numbers;
- `nat list`: finite lists of natural numbers; and,
- `nat => bool`: functions from natural numbers to truth values.

For more details see the reference manual *Isabelle's Logics: HOL* [11] in the Isabelle distribution.

Datatype Declarations starts up a dialog box to define the datatypes. Datatypes are the Isabelle version of familiar constructs such as enumerated data types in programming languages, Z free-types, or standard datatypes in ML. Any Isabelle datatype can be entered. An example is the simple enumerated type `Colour` introduced in the tutorial in Section 4.5 below. A more general example is the declaration of `'a list`, finite lists with elements of arbitrary type `'a`.

```
'a list = Nil | Cons 'a ('a list)
```

Thus, the list is defined iteratively to be empty (the element `Nil`), or obtained from it by prepending elements of type `'a`. For more details see the reference manual *Isabelle's Logics: HOL* [11] in the Isabelle distribution.

Constant Declarations starts up a dialog box to define the constants. Each constant[9] must be given a type – one of the available Isabelle/HOL types, or one of the datatypes or type abbreviations previously declared. The corresponding rules defining the function are entered in the `Rule Definitions` item of the `Definitions` menu (as discussed below).

[9] Here the word "constant" is used in the programming language sense – the constant can have any type, it simply doesn't change during the evolution of the state machine. That is, it has no dependence on the configuration. This is as opposed to a "variable", whose value changes depending on the configuration.

Other Declarations starts up a dialog box to define any other ML code that a user may wish to be added to the .thy file. No intermediate parsing will be carried out on this code, so errors will not be picked up until the state machine theory is being loaded (which occurs during compilation). This is an option for the advanced user; i.e., someone who is familiar with the use of both ML and Isabelle. To "register" as an advanced user, the variable 'advanced' in `configs/config.tcl` (or the user's own copy of config.tcl in `$HOME/isabelle/etc`) should be set to 1.

Rule Definitions starts up a dialog box to enter the named rules which define the constants declared in the `Constant Declarations` window. It is usual practice in Isabelle/HOL that these rules are simply definitional rather than axioms which extend the underlying logic. This is expected, though not enforced, in DOVE.

Heap Declarations starts up a `Heap Variable Declarations` dialog box to define the heap variables. Each variable must be given a type, one of the available Isabelle types or one of the datatypes previously declared.

Input Declarations starts up an `Input Variable Declarations` dialog box to define the input variables through which the state machine communicates with external entities; i.e., the inputs of the environment to the state machine. Each input must be given a type, one of the available Isabelle types or one of the datatypes previously declared.

Transition Definitions starts up a dialog box to define the transitions which are associated with the edges of the state machine. The definition of a transition consists of the condition under which the transition occurs, and what happens when the transition occurs. This is explained in more detail in Section 4.3. There are a number of points worth noting here.

- Each edge on a state machine graph must have a transition associated with it; however, several edges can have the same associated transition. A given edge label is actually the name of the associated transition. Thus, it is important to keep the following in mind when renaming edges/transitions:
  - If several edges have the same label, then a change in the transition definition (via the editing window of the dialog box) will affect all of them.
  - Similarly, if an edge is renamed (as discussed in Section 4.1.3), then the transition definition of the old label is no longer used in the state machine. If the new name coincides with that of an existing transition then this is the transition now associated with the renamed edge. Otherwise, the transition so labelled is not yet defined.
  - The method of renaming the actual transition in a given state machine is explained in Section 4.3.
- Selecting a transition in the dialog box's list of defined transitions causes all edges associated with that transition to be highlighted.
- Conversely, double-clicking on an edge of a state machine graph will bring up the editing window of the corresponding transition, or for a new transition if none has been created yet.

Property Definitions starts up a Properties Manager dialog box to define the properties which are to be proved in the analysis of the state machine in a later proof mode session. As mentioned earlier, this is the only declaration window which is available in a mode other than Edit mode – it is available in the proof mode. It is also special in that it has an Options menu.

The Properties Manager is accessible in Edit mode simply for inputting the desired state machine properties – there is no direct interaction with the Prover available in Edit mode. This "properties" view of the state machine can be very convenient for the formal design, where a subset of properties can play the role of formal specification for the machine. Thus, having these available before any state machine analysis is attempted is useful. Here it is worth just noting that the functionality as a declaration window interaction is essentially the same as all the others discussed above. However, to utilise properties effectively the user needs more information. To this end, Chapter 6 gives considerably more detail on the use of the Properties Manager window. Moreover, properties per se are also discussed elsewhere: a general discussion on state machine properties in Section 2.7; there is a discussion of how to go about formulating properties in Chapter 7; the exact syntax for properties is presented in Appendix B.

Include Theories brings up a very simple text-entry window in which the user records the Isabelle theory files on which the state machine theory depends. As stated on the title bar of this text-entry window, the theory files are to be entered separated by (at least one) white space.

Like the Other Declarations option above, this is a feature included for the advanced user. The theories to be included may have been built earlier by the user, or taken from some public/private external source, or be part of the Isabelle/HOL source directory of theories. They are typically theories which include theorems to allow reasoning about objects in the state machine theory. An example would be a theory of a timing-clock, which could be included in an elaboration of the TrafficLights example; or a theory of encryption/decryption which could be included in a state machine analysing security protocols. A more pedestrian example is the theories of real numbers in the Real subdirectory of the HOL source directory of the Isabelle distribution. In this last example, one would type the absolute path name of the required file; eg,

```
$ISAHOME/Isabelle99/src/HOL/Real/RealDef.thy
```

where $ISAHOME denotes the appropriate directory in the user's system.

4.2.5 Other displays on the menu bar

Between the Definitions menu and the mode buttons – which have been discussed in Section 3.3 – is the grid size display, which shows the current grid size. When clicked on with mouse button 1 a pull-down menu containing some of the more commonly used grid sizes is displayed, thus allowing the user to quickly change grid size.

Finally, the Help menu contains three options:

- **About:** displays an informational/welcome window for DOVE.
- **User Manual:** starts up Acrobat – if it isn't already running – and displays the PDF form of this user manual.

- **Logging:** displays logging messages, which can be used as a report to the development team if DOVE exhibits any unusual behaviour.

4.3 Transitions

As mentioned in the last section, each edge on a state machine graph corresponds to a transition of the state machine. The definition of a transition in DOVE consists of three parts; namely, in the order they must appear if used in the definition, the *Let* declaration, the *Guard* (or precondition), and the *Act* (or action list). Their use is explained in the following subsections. A Let declaration is optional, either or both of the others can also be omitted, but the choice corresponds to a specific declaration statement. The elemental words (variable names, and those making up the expressions in the different transition definition parts) have the syntax summarized at the beginning of Section 4.4.

4.3.1 The Let declaration

The Let declaration introduces local variables – i.e., variables only defined inside the current transition – to abbreviate expressions in the Guard and Act sections of the current transition definition. This is clearly an optional part of the transition definition. An example of a Let declaration is:

```
Let: temp1 ← (x + y);
     temp2 ← (x − y);
```

The `Let` keyword identifies the beginning of the Let declaration. The keyword is then followed by any number of local variable declarations. Note that having each assignment on a new line is not required, however the semicolon assignment separators and terminator *must* be written.

The ← is the assignment operator. On its left-hand side is the name of the new local variable, and on the right-hand side is the expression defining the value of the local variable. This expression can depend on the names of local variables defined earlier (i.e., higher in the Let declaration – thus ordering of the assignments can be important. It can also depend on *any* of the variables or constants defined in the state machine theory. The only restriction is that the resulting expression have the correct type for the local variable to be used as desired.

4.3.2 The Guard declaration

The Guard (or precondition) is a Boolean expression (predicate) over the memory, including any variables defined in the `Let` declaration part. It sets the condition under which the transition is enabled. An example of a precondition is:

```
Guard: (x = 0) or (y < 1)
```

The Guard keyword identifies the start of the precondition. DOVE will accept definitions where the precondition is omitted (i.e., the entire Guard statement including the keyword is omitted), it simply assumes that such transitions are enabled, and hence are equivalent to the precondition

```
Guard: True
```

The usual logical operators are available for use in the definition of the precondition. The notation is: =, <, neq, and, or, Not, Implies. Temporal operators cannot be used for transition definitions in DOVE state machines.

4.3.3 The action list declaration

An action list is a semicolon-separated and semicolon-terminated list of assignment statements which are applied in parallel: the order of assignments is immaterial, and all occur at the same time. It is an optional part of the transition definition. An example of an action list definition is:

```
Act: x ← y;
     y ← 1;
```

The Act keyword identifies the start of the action list declaration. The keyword is followed by any number of heap variable assignments separated and terminated by semicolons, the syntax being similar to that for the Let declaration (however, the ordering of the assignment statements is irrelevant as mentioned above). In particular, the assigned expression can depend on *any* of the variables (including local variables defined in the Let declaration) or constants defined in the state machine theory. The only restriction is that the resulting expression have the correct type for the assignment to the given heap variable to make sense.

DOVE will accept a definition where the action list is omitted (i.e., the entire Act statement including the keyword is omitted), it simply assumes that no change of state is intended. This would be made explicit as

```
Act: Skip;
```

4.3.4 Editing, deleting and renaming transitions

A transition which corresponds to a particular edge can be edited or created by double-clicking on the edge.

To delete a transition, it is first necessary to delete all corresponding edges from the state machine graph as described in Section 4.1.3. Having done this, the transition definition can be deleted by bringing up the dialog box of the Transition Definitions option of the Definitions menu. Selecting the undesired transition, it is removed by clicking on the Delete button. If the edge is not deleted first, then DOVE will simply respond by stating that the transition is in use.

To actually rename a transition in a given state machine one should not simply relabel the edge. Rather, the renaming should be done in the Transition Definitions option of the

Definitions menu. Once a transition has been renamed in the declation window, it will also be renamed on the state machine graph.

To summarise, rules concerning the editing and renaming of edges on the graph are as follows:

- When the user double-clicks on an edge on the graph, the Transition Editor appears, either for a transition of that name, or for a new transition of that name if no transition has yet been declared. This is the same Transitions Editor that can be invoked from the Transition Definitions Window.
- If the user renames the transition using the Transition Editor, all the edges which were associated with that transition will also be renamed automatically, in addition to the currently selected edge.
- To associate an edge with an existing transition, the user should edit the name directly on the graph (i.e. without the help of the Transition Editor).
- If the user double-clicks on an edge in order to invoke the editor, and renames it to an existing transition, this should produce an error, since if an attempt is made to do this for an edge which is not yet associated with a transition, the definition and description for the existing transition will be lost.

Note that, due to a bug in the underlying Tk graphical mechanisms, it is possible to edit a label on the graph without it actually being ”in focus”; i.e., the user will be able to pass the mouse over the edge (which toggles the colour to the selected shade), move the mouse slightly so that the edge appears to be unselected, and may find that the cursor is still waiting for input within the label. However, this feature is merely an oddity and should not cause any problems during state machine input.

4.4 Mandatory elements of state machine design

There are a number of requirements for specifying a state machine in DOVE so that the animation and formal verification implementations will be consistent. The corresponding rules and checks are all very simple. However, DOVE does not enforce them all automatically, but rather provides checking mechanisms for identifying fatal violations. Many local syntactic errors will be identified by the various data entry boxes and global syntactic errors are identified by the menu item Check/Compile. A description of the possible diagnostic messages is given in AppendixD.

The user should be careful to follow the rules described in the following subsections during the construction process.

4.4.1 The identifiers

All names in DOVE are strings of alphanumeric characters and underscores, which must begin with a letter. The full syntax for defining DOVE state machines is further explained in AppendixB.

4.4.2 Rules for initialisation of the state machine

There must be an initial state chosen. The choice is entered via the window invoked by the option `Initialisation` in the `Definitions` menu.

4.4.3 Naming rules

M1 Nodes must have distinct names. This is due to the fact that the names of nodes represent the states of the machine.

M2 Each type, datatype, constant, and variable must have a distinct name.

M3 The name `transition` is reserved and may not be used to name a type, datatype, constant, or variable.

Violations of **M1**, **M2**, and **M3** are identified by the `Check/Compile` operation.

4.4.4 Declaration and type rules

M4 Each (non-standard) type abbreviation appearing in a constant, or variable declaration must be declared.

M5 Each constant, or variable appearing in a transition definition must be declared (the syntax for transition definitions is discussed in Section 4.3).

M6 Each constant, or variable appearing in a transition definition must obey the type rules for its declared type.

4.4.5 Check assignments

M7 No input variables should be assigned to in the transition action.

M8 No local variable, introduced in a transition's Let declaration, should be assigned to in the transition action.

M9 No heap variable should be assigned to twice in the same transition action.

In the action list of a given transition definition in the DOVE state machine model, heap variable assignments occur in parallel. DOVE does not enforce the distinctness of assigned variables: if a heap variable is assigned to more than once in an action, then the DOVE tool will choose one assignment or the other, but the designer has no way of predicting which (except through animation or proof).

4.5 Tutorial: construction of TrafficLights

In this tutorial the user will construct a few of the possible states and their linking transitions in the TrafficLights state machine, which models traffic lights at a N/S and E/W intersection. To profit from it, the reader should have at least scanned Chapter 2. If at any stage there is some confusion with the tutorial, or an apparent error, the user should easily be able to rectify the problem by comparison to a full TrafficLights session which may be started independently as described in Section 3.1.

It is assumed that the reader has the DOVE tool appropriately set up as in Chapter 3, and is in a working directory `TrafficLights` – which includes the file `TrafficLights.smg` – as discussed in the preamble to Section 3.2. In the following, the command-line prompt is denoted by the `>` at the beginning of the command line. Also, in common with the rest of the manual, grammatical notation will not be included in command lines.

Open a new DOVE session. As explained in Section 3.1, this is done by typing

```
> dove
```

A blank, grey, grid-patterned canvas will appear on the screen. This is the DOVE state machine window which is used for designing the machine. The user will notice that the “pencil” icon on the menu bar is blue, whereas the other icons are green. This indicates that the DOVE session is in editing mode.

The user will now design the first three states in one of the cycles of the TrafficLights example, which requires just a subset of the attributes needed in the more realistic model. First the topology of three nodes connected in a line, with appropriate labels, is constructed. Then some datatypes are introduced to model the lights – not surprisingly, these will be the colours of the lights and the directions of the intersection. A constant is required to denote the maximum number of cars which can be waiting in any given direction after which the lights must change. It has the type of natural numbers. Also, a heap variable must be introduced for each machine attribute of interest. In this session, the attributes of interest will be a variable which encodes the colour of a given light, and a variable which encodes which direction was last green. Input variables are introduced to encode the environment effect. In this session these will be the number of cars waiting at each of the north and south lights when they are red. They have the type of natural numbers. Finally, the initializing transition requires that all lights are red, and the next transition will correspond to the E/W lights changing to green.

4.5.1 Topology

The first step in the design is to construct the topology of the machine. This will involve the creation of two nodes descending in an evenly-spaced line from near the middle of the canvas, followed by the creation of edges corresponding to transitions, connecting those nodes. This is done as follows.

- Click with mouse button 1 near the centre of the canvas. A node will appear, with the default label `Node`. Notice that the node has a radius of one grid unit. At this point, the user has already constructed the topology for a valid (although not very interesting) state machine.

- Similarly, create a second node, about eight grid units apart on a direct vertical line down from the first node.
- Now click and hold mouse button 1 at (just inside) the bottom of the first node, and drag the arrow which appears down to (just inside) the top of the second node. On releasing the button, the edge just created will snap to the grid, with the default label `Edge`.

At this point, the state machine should appear as shown in Figure 4.2. Note that a node has four "vertices" to which edges can be attached, which appear at the "compass points" of the circle. In this language the south vertex of the first node has been joined to the north vertex of the second node.

If another node or edge was accidently created, then it may be deleted as follows.

- Put the mouse cursor on the offending node or edge (create one if there isn't already one).
- Holding the Control key down, click with mouse button 1. Note that, before clicking, the mouse cursor turned into a "skull-and-crossbones". After clicking, the object is deleted.

4.5.2 Moving graph objects

If the layout of the graph is not as desired, the nodes and edges can be moved. Move the second node to the same height as the first node and about eight grid units to its left as follows.

- Put the mouse cursor on the second node.
- Click and hold mouse button 2. A red shading will appear around the other nodes and the edges. This denotes the region into which the second node *cannot* be moved.
- Drag the second node to the required position about eight grid units directly left of the first node, and release the mouse button. Notice, as shown in Figure 4.3, that the joining edge snaps to the grid, but its start and finish are unfortunate in that they do not allow it to take the shortest path.
- Click with mouse button 2 on the head of the arrow on the north vertex of the second node, and drag it to the east vertex.
- Drag the tail of the same arrow from the south vertex to the west vertex of the first node.

4.5.3 Labelling graph objects

Now the state and transition labels will be assigned. Label the top central node (the first node) as `AllRed`, the node to the left of it as `EWGreen`, and the joining transition as `EWChange`. This is done as follows.

- Move the mouse pointer to be inside the node whose name is to be modified. A text-insertion cursor will now blink at the end of the label. Delete the default label `Node` and replace it with the desired label, using the keyboard.

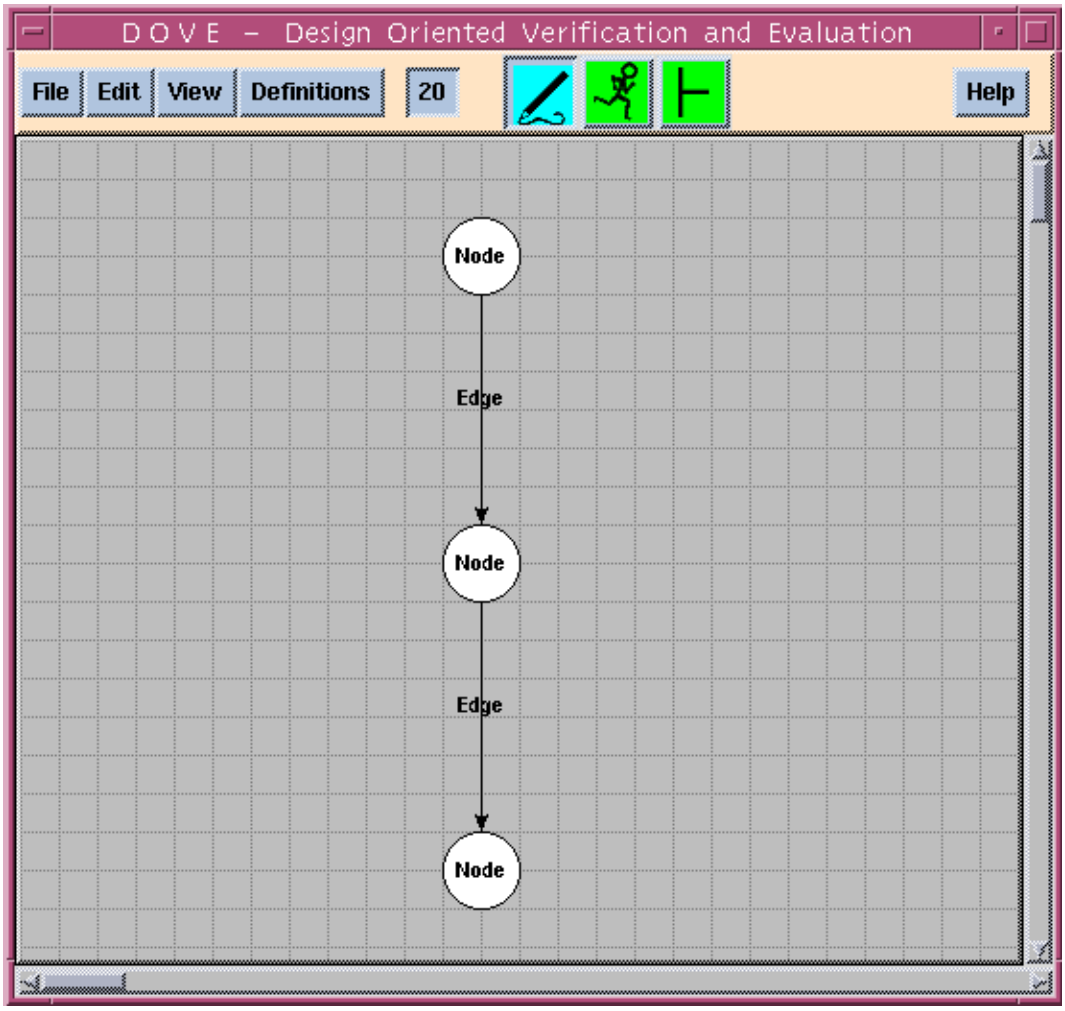


Figure 4.2: The state machine segment initially.

- Relabel all the nodes to the names suggested above.
- Now move the mouse pointer onto any part of the edge leading from `AllRed` to `EWGreen`. This edge and its label will turn blue, and the blinking text-insertion cursor will appear. Change the default label `Edge` to `EWChange`.

The machine, which should appear as shown in Figure 4.4, is now ready for definition.

4.5.4 Machine definition

Now the machine attributes must be declared and the transitions defined, for which purpose the options under the `Definitions` menu button are used. Our theory will use just the base types of Isabelle, so ignore the `Type Declaration` option. Declarations must be given in several of the remaining options to reproduce the attributes discussed above.

Before proceeding, it may be useful to realize that declarations (for any of the declarations

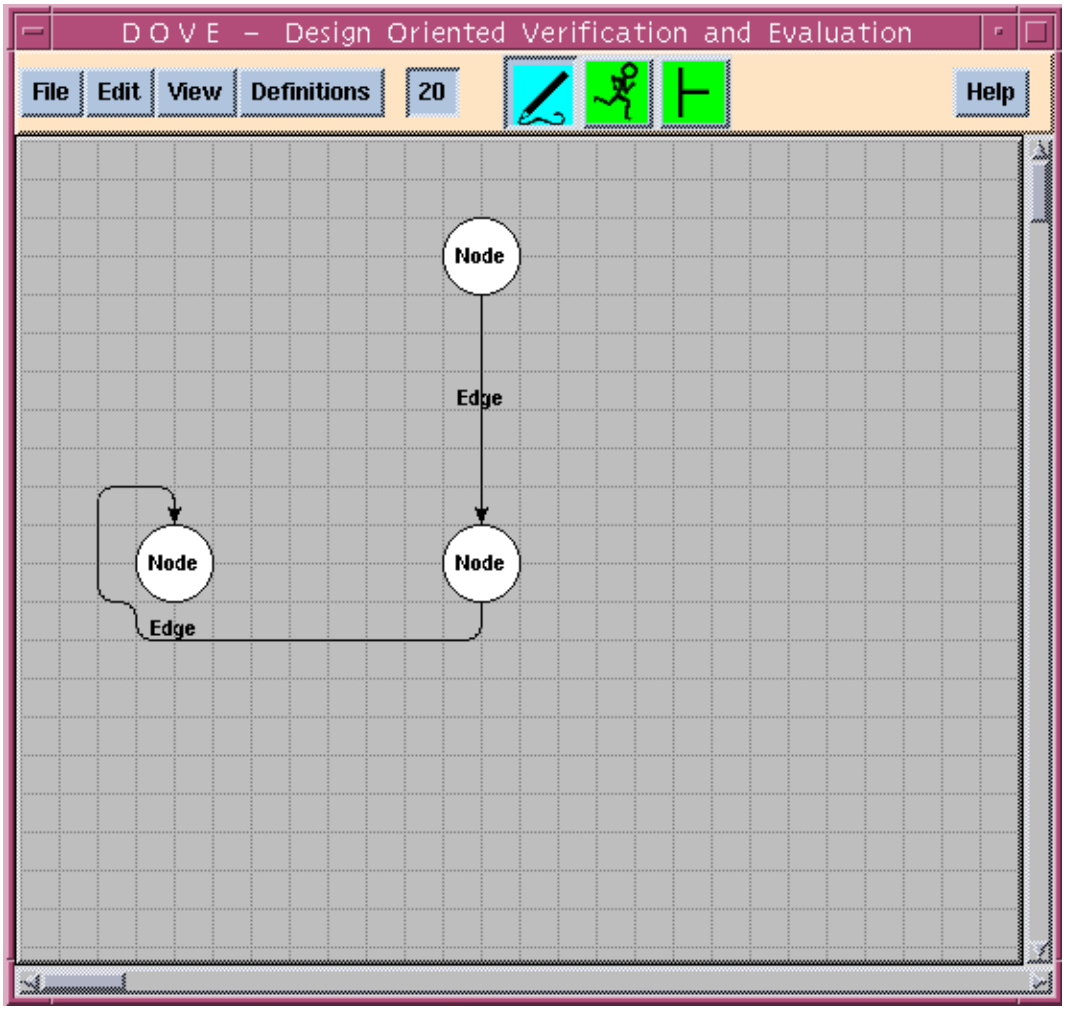


Figure 4.3: The edited state machine segment.

windows brought up by the options in the `Definitions` menu) are displayed differently, depending on whether they have been parsed correctly, parsed and found to be invalid, or not parsed at all:

- If the declaration has not been parsed since its last modification, its name and status will be displayed in black italics, and its status will be listed as “UNCHECKED”. In this case, the user will be obliged to commit the declaration prior to compilation, animation or proving.

- If the declaration has been parsed and was found to be correct, its name and status will be displayed in blue, plain font, and its status will be listed as “OK”.

- If the declaration has been parsed but was found to be invalid, its name and status will be displayed in red, bold font, and its status will be set the relevant error message.

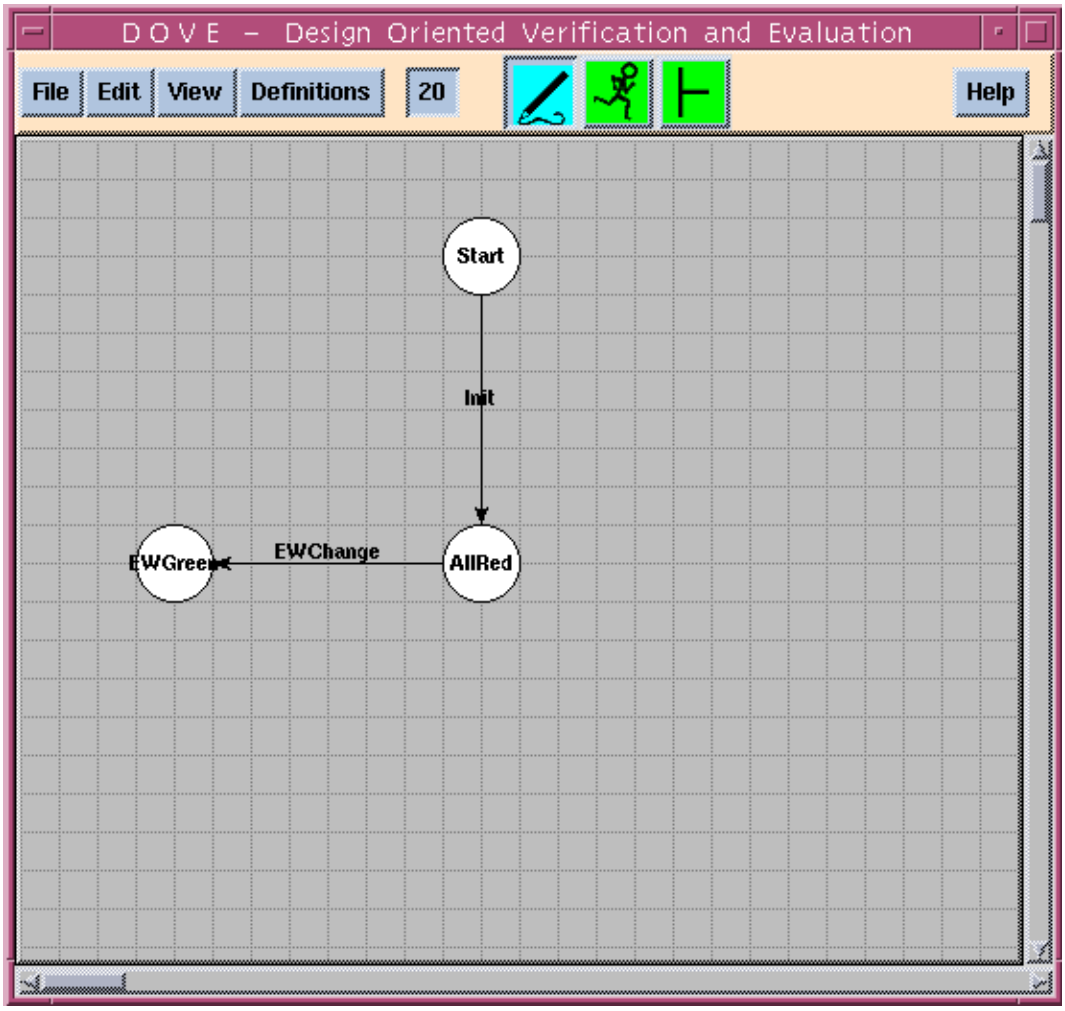


Figure 4.4: The labelled state machine segment.

4.5.4.1 Datatypes

Declare two datatypes: `Colour`, which can be `Red`, `Amber` or `Green`; `Direction`, which can be `EW` or `NS`. This is done as follows.

- Pull down the `Definitions` menu and select `Datatype Declarations` (selection will always be done with mouse button 1), to bring up the declaration window.
- Click on the button labelled `New`, causing an empty editor window to appear.
- Put the mouse cursor in the `Name` text-entry field of the dialog box which appears, and type `Colour`.
- Since `Colour` will be a simple enumerated type, there is no input needed in the `Parameters` text-entry field.
- Type `Red | Amber | Green` in the text-entry field labelled `Definition`.

- A comment may (and should) be inserted in the bottom text-entry field.

- Click on the button marked Commit in the editing window. If there is an error in the definition, the editing window will remain visible, and the error will be reported to the user. If the declaration has been entered correctly, the new datatype will be accepted, and will be displayed in the Datatype Declarations Window blue with a plain font.

- Similarly, enter the datatype Direction and its allowed types as suggested above.

- Click on the Close button at the bottom right to close the Datatype Declarations Window.

4.5.4.2 Constants

Declare the constant MaxCars of type nat. Here nat is the Isabelle base type corresponding to natural numbers. Do this as follows.

- Select the Constant Declarations option of the Definitions menu, then select New and insert the name MaxCars in the dialog box which appears – all as before.

- Type nat in the text-entry field labelled Definition. Again insert a comment if desired, click on the button marked Commit in the editing window, and then Close.

Note that MaxCars, the maximum number of cars which can be waiting in any given direction after which the lights must change, could be given a particular numerical value via a named rule. Let's set MaxCars to be 3 as follows.

- Select the Rule Definition option of the Definitions menu, then select New and insert the name MC_three in the dialog box which appears. This is the name of the rule.

- Type MaxCars = #3 in the text-entry field labelled Definition. Again insert a comment if desired, click on the button marked Commit in the editing window, and then Close.

Another way of dealing with the traffic lights situation would be to give preferential treatment to the NS direction. To do this we would introduce instead a constant FMaxCars of type (Direction => nat). We could then have the rule FMaxCars_NS which sets (FMaxCars NS) to 1, and the rule FMaxCars_EW which sets (FMaxCars EW) to 3. Alternatively we could have a single rule, FMaxCars_def, with definition (note that % is the Isabelle syntax for lambda-abstraction, while the if-then-else is Isabelle HOL syntax)

```
FMaxCars = (% d . (if d = NS then #3 else #1))
```

This is a good example to show that the word "constant" should be taken in the programming language sense!

4.5.4.3 Variables

Declare some of the heap and input variables of the traffic lights system, in the same way as declaring constants:

- select Heap Declarations to declare the heap variables ELight, WLight, NLight, SLight – all of type Colour – and the variable LastGreen of type Direction.
- select Input Declarations to declare the input variables NCars and SCars of type nat.

4.5.4.4 Initialisation

An initial state for the state machine executions must be defined. In this case, it is the state called AllRed. Select Initialisation from the Definitions menu, enter the name of the initial state, and the initial condition:

```
(NLight = Red) And (SLight = Red) And (ELight = Red) And (WLight = Red)
```

4.5.4.5 Transitions

Since an edge named EWChange has already been defined, a corresponding transition of the same name needs to be created. The definition for this transition is:

```
Guard: LastGreen = NS

Act: ELight ← Green;
     WLight ← Green;
```

The definition can be entered as follows:

- Pull down the Definitions menu and select Transition Definitions.
- Press the button marked New, which will result in an empty transition editor window appearing.
- Put the cursor in the text-entry field of the window labelled Name, and type EWChange.
- Now type the definition – Guard: etc, precisely as it appears above – in the text-entry field of the editing window labelled Definition.
- A comment should be inserted in the bottom text-entry field.
- To check that the definition has been entered correctly, click on the Commit button of the editing window. If there is an error in the definition (note that the semicolons in the Act statement must be written!), the editing window will remain visible, and the error will be reported to the user. If, however, the definition parses correctly, the editing window will be dismissed. Alternatively, the user may choose to parse the declaration at a later date, in which case the Close button should be used to close the editor window.

The editing window for a specific transition can be invoked by pressing the button marked Edit on the declaration window, or by double-clicking on the name of the transition, or by pressing the Return key when the transition is selected, or by double-clicking on the corresponding edge of the state machine graph.

4.5.5 Renaming a transition

The user is now well on the way to designing a state machine model of traffic lights. Indeed, the state machine could be compiled at this stage, by selecting the Check/Compile option of the Edit menu.

However, there is one obvious problem with the model – the name of the transition. Certainly a full model would need transitions where the E/W lights turn amber, and red. So, the second transition should be distinguished from these by a name such as EWChangeGreen. To make this change:

- Select Transition Definitions from the Definitions menu, select EWChange from the dialog box which appears, and rename the transition using the editor window.

Change the name of the transition to EWChangeGreen using this approach. Notice that all instances of the transition name have been changed appropriately, even the label on the state machine graph.

Note: the user may think that the state machine graph can be edited directly, as explained in Section 4.5.3, by simply relabelling the second edge to the desired name. This is incorrect. To see this, change the edge label (which should now read EWChangeGreen) to EWChangeGreen1 directly on the graph, and then double-click on the newly-relabelled transition to find that there is none such which has been defined! Moreover, select Transition Definitions from the Definitions menu, and see that the name of the defined transition has *not* been changed. So, invert this procedure to restore the label to EWChangeGreen.

Finally, compile the state machine by selecting the Check/Compile option of the Edit menu.

4.5.6 Saving and reloading the state machine

Having gone through this exercise (hopefully feeling that in fact it was remarkably easy!) it is nice to save and appreciate the result. To save the state machine which has been produced, select the Save As option from the Edit menu. Select a name 0.smg – where 0 does not already appear in the working directory – such as (this name will be assumed below) TLSegment.smg.

The file TLSegment.smg now encodes the state machine segment defined above. Thus, for example, it can be loaded directly to begin a new session. Do this by selecting the Load File option in the File menu, and then select TLSegment.smg in the dialog box which pops up. Notice that the file is loaded without asking for confirmation. Alternatively, the user could first clear the current session – and so restore the blank canvas – by selecting the New option in the File menu, and then reload the file as just described. User confirmation *will* be required if the current state of the state machine graph is not saved – to see this, delete the “d” in the label AllRed and repeat the

above operations (press `Cancel` on the dialog box). Now, to restore the state machine graph to its previously-saved state, choose the `Restore` option in the `File` menu (and this time press `OK`).

To savour the results the user should select the `Create Documentation` option in the `File` menu. If the machine compilation succeeds a pdf documentation file for the state machine design will be produced automatically. If the compilation fails the user will be prompted with the errors, and asked whether the documentation should be created. If so, the design to date – including the error messages – is recorded in the pdf file.

Chapter 5

Animation

The Animator displays the change in the values of the state machine's variables along a chosen execution path. In DOVE, animations for a given state machine are carried out directly on the corresponding state machine graph. The user selects a start state for the animation by clicking with mouse button 1 on the desired node, and proceeds step-wise through clicking similarly on a chosen path of intermediate edges to the desired finish. DOVE has a convenient colour scheme, discussed below, that visually aids understanding. The chosen start node does not have to be the initial state of the state machine initialisation!

DOVE also allows the user to store animations, which may then be saved to disk along with the rest of the state machine parameters, to be retrieved in later DOVE sessions. It is also possible to load an animation from a text file written in a specific format - this will enable DOVE to pick up animated sequences produced by other processes.

The "hands-on" reader may wish to move quickly to the tutorial session in Section5.4, referring to earlier sections when required.

5.1 DOVE window display in animation mode

The Animator is entered by clicking on the "runner" (animation mode) button (see Figure 3.2(b)) of the DOVE state machine window menu bar. This brings up two windows, as seen in Figure 5.1 and Figure 5.2, in addition to a new set of graphical objects attached directly to the base of the state machine graph. The latter are known as the Animation Controls. The use of the two windows and the Animation Controls is explained in the following subsections.

Before the change to animation mode is actually enacted, DOVE automatically calls the Check/Compile option (from the Edit menu of the state machine graph window). It is not possible to start an animation of a state machine theory with fatal errors in the ensuing diagnostics. If no fatal errors are found, the corresponding diagnostic window will automatically disappear in a few seconds.

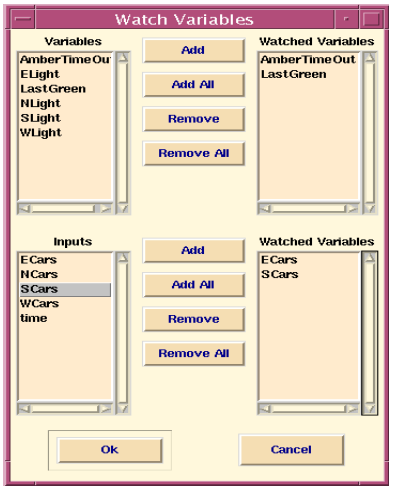


Figure 5.1: The Watch Variables window.

5.1.1 The Watch Variable window

The `Watch Variables` window (see Figure 5.1) is used to select the variables to be tracked during the animation. It contains two sets of lists - one for the heap variables and one for the input variables. The lists on the left contain all the variables defined for the current state machine, while those on the right contain those selected to be watched during the animation. The buttons located down the centre of the window are used to add and remove variables from the right hand "watched" lists. Note that addition or deletion can also be achieved by double-clicking on the name in the appropriate window. In the example session from `TrafficLights` shown in Figure 5.2, the heap variables `AmberTimeOut` and `LastGreen` have been added to the watched list.

5.1.2 The `Animator` window

The `Animator` window displays the information of the current animation. The chosen start state is shown as well as the current state along the chosen edge path, colour-coded to the graphical display as will be explained below. The `Heaps` box contains the list of heap variables selected in the `Watch Variables` window, and their value in the current state of the animation is displayed opposite in the `Values` box. Likewise, the `Inputs` box shows the list of currently selected input variables, and their values. The `Path Condition` box shows the accumulated conditions from the transition preconditions along the chosen edge path, as explained further in the next subsection. Finally, a `Description` area is provided at the base of the window, so that users can keep track of their various animations by providing comments to be associated with them, in addition to descriptive names. A comment is entered in the `Description` frame via the dialog box invoked by the `Store` option under the `File` menu of the `Animator` window – as explained in the menu decription below.

The user also has the option of specifying values for variables:

- only *initial* values may be specified for heap variables, whose values in later steps of the animation are derived from the transition actions;

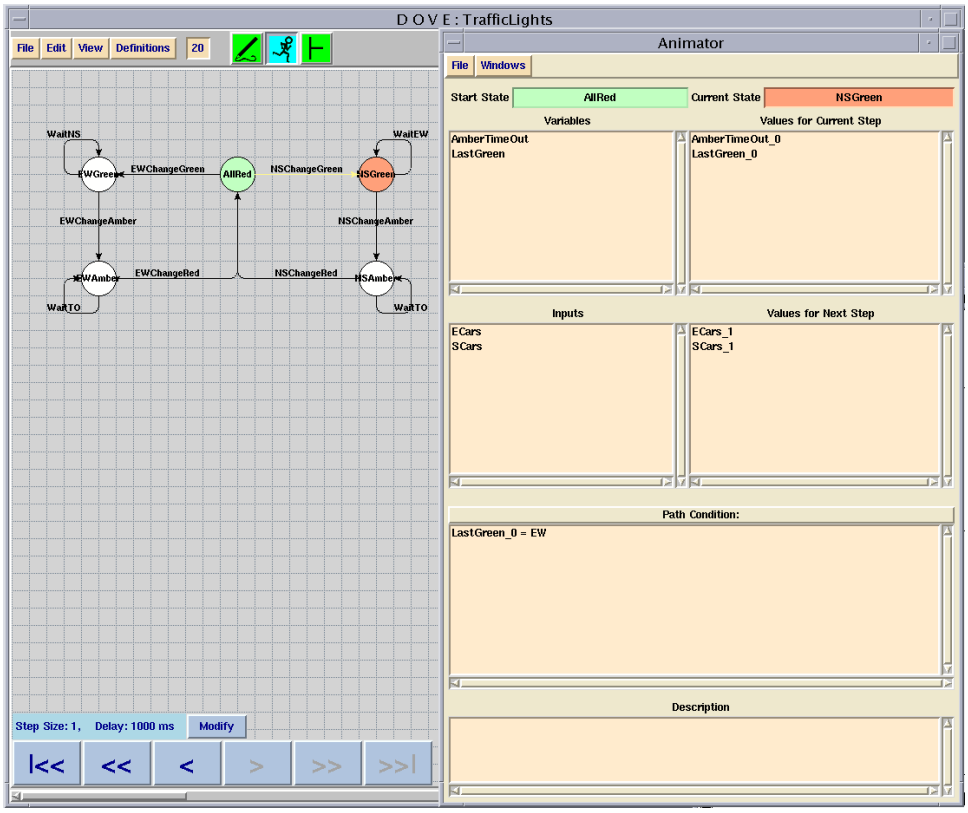


Figure 5.2: DOVE display after one step in an animation.

- insertion of initial values for heap variables must be done before proceeding to define a new animation – in particular, before the choice of the start state;
- input values can be specified at any stage – input variables model changes in entities external to the state machine, and thus may be modified at any point during the animation.

To insert the desired value the user need simply double-click on the value of the corresponding variable, type the required value into the dialog box which is invoked, and press the `OK` button. At this point the new value will be subjected to some superficial parsing. Note that numeric values must be prefixed with the # symbol, as this is the way they are recognized by the Isabelle proof tool.

The "watched" variable list can also be changed at any stage, as explained further in Section 5.3 along with the other features of the `Animator` menus.

5.1.3 Path conditions in the `Animator` window

The `Path Condition` box shows the accumulated conditions from the transition preconditions along the chosen edge path. These conditions determine whether the chosen path is a possible trace of the state machine execution; i.e., for the state machine to be able to execute the chosen sequence of transitions from the start state to the current state in the animation. If so, then all the

predicates listed must evaluate to true in the start state of the animation. The predicates are listed in order - thus they are added to the end (bottom) of the list in an animation step.

The Animator applies some logical simplification rules to the accumulated path condition. In particular, if the user attempts to activate a transition whose precondition fails, the path condition will evaluate to false, and as such will be displayed in the Path Condition box as False.

5.2 Animation via the state machine graph

The animation is driven by mouse operations, directly on the state machine graph drawn on the canvas of the DOVE state machine window. The state machine graph, at any given moment in the animation, will have a green node signifying the start node of the animation, and a red node signifying the current state of the animation (that which the animation has reached), with all nodes and edges in-between coloured amber. (Note, however, that all of these colours can be changed via the parameter file, config.tcl.)

5.2.1 Starting the animations

An animation is initiated by clicking with mouse button 1 on a chosen node. (Initial values for heap variables must have been entered prior to this, as discussed above.) That node is automatically registered as the start (and current) state of the new animation, which DOVE indicates by colouring it red. This fact is mirrored in the Animator window, in which both the Start State and Current State boxes will display the selected start node. The ability to select any node as the start implies that animations can be partial, spanning any consecutive sequence of edges of the state machine. Suppose EWAmber is registered[10] as the start node in the TrafficLights example – the beginning user may wish to try this while reading now. Note that, having done this, the value of ELight is not registered immediately – although it "must" be Amber in the chosen state. This is a design choice: DOVE shows the values as blank until they are first modified in the course of the animation. Indeed, an unusual choice of initial value for ELight in the start state would have overridden any intuitive understanding of what the value must be.

5.2.2 Animation

The user proceeds (forwards) through the state machine graph by clicking on an *edge* leading out of the current state with mouse button 1. DOVE then updates the information in the Animator window to reflect the chosen animation step. *After* this step, the current node – i.e., that where the chosen edge ends – is coloured red, the start node becomes green, while all intermediate nodes and edges are coloured amber. Clicking on an edge that does not leave the current node has no effect. The beginner may now wish to continue the animation begun above: entering values for watched input variables, and observing the predicates which build up (added successively to the *end* of the Path Condition list).

[10] DOVE requires user confirmation if a new animation is started while another is in progress.

5.2.3 The Animation Controls

As described above, the Animation Controls are graphical objects which are attached to the base of the state machine graph during Animation mode. They consist of a set of control buttons, together with a display area listing the current values for the `delay` and the `step size`, in addition to a `Modify` button which invokes a dialog window allowing the user to update those values.

The `step size` can be any integer in excess of 0; this is merely the number of transitions that will be traversed whenever the user steps forwards or backwards in the animation. The `delay` is concerned with automatic animation; i.e. when the user chooses to move either forwards or backwards continuously through the animation for as far as the current animation will allow. The `delay` specifies the minimum number of milliseconds that each step in the animation should take in real time, the objective being to give the user sufficient time to examine the results of each step. The usefulness of this parameter will be more apparent with a fast computer, for which the delay would specify the actual time that a single animation step should take (rather than a minimum time).

At the base of the controls is a series of operational buttons, which are used in order to traverse a pre-defined animation. The outermost buttons, marked |<< and >>|, indicate that the animation should proceed directly to either the beginning (i.e. the start state) or the finish, respectively. The innermost buttons, marked < and >, are synonymous with the operations “Undo” and “Redo” - they cause the current positioning of the animation to move either backwards or forwards. In general, the user will wish to move in single steps; however, the `step size` parameter can be configured to skip additional steps, as described above. The remaining buttons, << and >>, are for automatic animation; i.e., they allow the animation to move continuously either backwards or forwards, for as long as the currently defined animation will allow. The `delay` parameter dictates the speed of the animation, as discussed earlier.

While any of the operational buttons are in action, the graph’s colour scheme will be updated to reflect each change in the position of the animation.

5.2.4 Named Animations

Thus far, the user has been working with a single current animation. Animations may, however, be stored within the state machine, with the constraint that each stored animation must have a unique name. The animation can then be selected at a later date for replay. This functionality is controlled by the `File` menu. Note that whenever a named animation is current, its name is displayed on the title bar of the Animator Window.

Stored animations may be adversely affected by changes made to the state machine during editing. For example, if an animation proceeds to the state x but, during editing, the state x is removed, the stored animation will no longer be valid, and errors would occur if an attempt was made to execute such an animation. To warn the user about invalid animations, and to ensure that those animations do not affect the use of DOVE, all stored animations are checked as the user enters Animation mode. Any animations which do not conform to the current configuration of the state machine are marked as invalid; likewise any animations which were previously invalid but have now been revived (due to a correction in the state machine design) will be available once more. It is, however, possible to correct invalid animations - this will be dealt with in a later section.

5.3 The menu bar of the Animator window

5.3.1 The File menu

New The New option clears the current animation. This has the same effect as clicking on the animation mode (”runner”) button during an animation session (as discussed in Section 5.3.3). Any resulting animation work will be anonymous - i.e., the animation cannot be stored until the user gives it a name. This is done by selecting Store under the File menu (see below).

Restore This restores the current named animation to the state in which it was last stored (see the Store option below).

Load File Sometimes it may be useful to feed the results of some other application to the DOVE animator. Allowing the user to load an animation from an external source will enable data which is output by other applications to be modified by the user via a standard editor, prior to presentation to the DOVE animator.

The format of a file containing an animation to be loaded is as follows (key words and symbols which must be included when appropriate are shown in bold):

> **Start State** startState
>
> **Transition Steps [** trans1**,**state1**,**trans2**,**state2 **]**
>
> **Initial Values [** heapVar1**:** val1**,** heapVar2**:** val2 **]**
>
> **Input Values [** inputVar1**: [**val3**,**val4**],** inputVar2**: [**val5**,**val6**]]**

White space is ignored, but the file should have no other contents. If the file is loaded successfully, its contents will become the current (anonymous) animation, positioned prior to the first transition.

Note that it will not be possible to save an animation on its own to a separate file - all animations that are associated with a particular state machine are saved in the state machine file, and there is no reason that they should continue to exist externally to the state machine.

Store Selecting Store invokes a dialog box which gives the user the opportunity of specifying a name under which the current animation should be stored. If the current animation already has a name, the entry field will display that name, which the user can either accept or else may enter a new name, if it is appropriate to keep a separate copy of the animation as it was last stored. The name given to the animation must be unique; if it is not, an error message will be displayed. The dialog also allows the user to enter a comment to appear in the Description frame of the Animator. This can be done at any stage, the accumulated comment is always shown.

Once the current animation has been named, its name will be displayed on the title bar of the Animator window. This name will be associated with the current animation, until a new stored animation is selected, or an animation is loaded from a file, or the current animation is cleared via the New option.

Note that an animation consists of the start state together with all the edges that lead to the finish, and the values for the variables at each step of the way. If the user has backtracked to a previous node in the animation prior to storing, the animation in its entirety will still be stored; that is, the current position within the animation is not considered to be relevant. If, however, the user backtracks, but then fires a different transition leading from that node, thus changing the course of the animation, the old branch of the animation will be discarded, and only the new branch together with the path that preceded it will be stored.

Note also that storing an animation does *not* imply that anything has been saved to disk - after storing an animation, the user should select the `Save` option under the `File` menu *of the state machine graph window* if the state machine and its stored animations are to be saved in a persistent manner.

Remove This option provides a submenu presenting the user with a list of all the currently stored animations – including those that are invalid for the current state machine. The user can remove a single named animation, or all the stored animations. This option has no undo - if the user makes a mistake at this point, the only option is to return to Edit mode, and restore the state machine file, since the removal operation only removes animations from the non-persistent state machine, and does not become permanent until the next save.

Note that it is possible to remove the current animation (if named). This will cause the current animation to be cleared, after which the current animation will be anonymous.

Select This option also provides a submenu listing all the currently stored animations; those which are invalid for the current state machine, however, are tagged as `Invalid`, so that the user cannot replay such an animation until it has been corrected. This is done by editing the text version of the animation which appears in a separate dialog when an invalid animation is selected by the user, along with a description of the first error that was encountered when parsing the animation (the error may not be apparent from the test of the animation, which refers to the current state machine).

Animation syntax is exactly the same as that required for animations which are loaded from file - please refer to the menu option `Load File` above. Once a valid animation has been selected, or an invalid animation has been corrected, the current animation is cleared (without waiting for confirmation from the user). The requested animation is then loaded, and positioned at its finish - i.e. the entire path of the animation is displayed on the graph. The Animation Control buttons may then be used to traverse the entire animation, although the user may also add to or modify the animation, if desired.

5.3.2 The Windows menu

Show Path brings up a window that lists, in order, the names of the transitions that have been undertaken in the current animation, from the start to the current state.

Watch Variables brings up the `Watch Variables` window, which has been explained in the preamble to this chapter.

5.3.3 Exiting Animation Mode

To leave the animation mode the user must click on either the edit mode or proof mode button of the DOVE state machine window. A dialog box then appears, asking if the animation should be cleared. If the user answers `No` then the current animation session is resumed. If the user answers `Yes` then the current animation session is cleared and closed (and the desired mode opened).

If the user clicks on the animation mode button during an animation session then a dialog box appears, asking if the animation should be restarted. If the user answers `No` then the current animation session is resumed. If the user answers `Yes` then the current animation session is cleared, but the `Animator` remains open.

Note that when the user leaves the animation mode, the currently named animation (if any) is remembered (in the non-persistent state). The next time the user enters animation mode this animation will be loaded automatically as the current animation under the assumption that the user will generally wish to continue working from the last known interruption.

5.4 Tutorial: animation of TrafficLights

In this tutorial the user will animate a few cycles of the TrafficLights state machine. To profit from it, the reader should have at least scanned Chapter 2; in particular, Section 2.1, which describes the various attributes and inputs of the traffic lights system.

It is assumed that the reader has the DOVE tool appropriately set up as in Chapter 3, and is in a working directory `TrafficLights` – which includes the file `TrafficLights.smg` – as discussed in the preamble to Section 3.2. In the following, the command-line prompt is denoted by the `>` at the beginning of the command line. Also, in common with the rest of the manual, grammatical notation will not be included in command lines.

Open a DOVE session with the TrafficLights state machine. As explained in Section 3.1, this is done by typing

```
> dove TrafficLights
```

The grey, grid-patterned canvas with the complete TrafficLights state machine will appear on the screen. This is the DOVE state machine window, which was used for designing the machine – as discussed in Chapter 4, and in the tutorial Section 4.5. The user will notice that the “pencil” icon on the menu bar is blue, whereas the other icons are green. This indicates that the DOVE session is in editing mode. Click on the “runner” icon which will then be the only blue one, indicating that the session is now in animation mode. The `Watch Variables` and `Animator` windows immediately appear.

If the TrafficLights version being used already has a stored animation, then the animator will come up referring to it, as discussed above. In this case, select the `New` option of the `File` menu on the `Animator` window. This clears the animation, and the user may now proceed as below.

From the `Heaps` list of the `Watch Variables` window, select the variables `AmberTimeOut`, `ELight`, `LastGreen`, `NLight`, and `SLight` to be watched. This is done as follows.

- Select `AmberTimeOut` by clicking on it once with mouse button 1 (selection will always be done with mouse button 1). Now click once on the `Add` button. (Alternatively, the user

can just double-click on AmberTimeOut.) This variable will now appear in the Watched Variables list. Similarly, select the remaining variables listed above.

- Click on the OK button, the Watch Variables window will disappear and the selected variables will be automatically inserted in the Heaps list of the Animator window.

Similarly add Inputs ECars and time to the Watched Inputs list (bring back the Watch Variables window via the corresponding option in the Windows menu).

The watched variables are automatically given default initial values in the Animator window. Change the initial value of LastGreen to be EW, as now described.

- Double-click on the *value* of the variable LastGreen. The value appears in the Values for Current Step frame, and is initially the default value LastGreen_0.

- Insert the desired value EW directly into the text-entry box which appears, and click OK. The desired value is then automatically inserted in the Animator window.

Now begin an animation from the AllRed state by clicking on the corresponding node of the graph with mouse button 1. Notice that the AllRed state is inserted as the Start State, and also as the Current State, in the Animator window. The colour of the node changes to red, since this is the colour for the current node, which takes precedence over the colour for the start node, green.

To proceed forwards in the animation, click once on the edge labelled EWChangeGreen, with mouse button 1. The current state (at this stage, EWGreen) will now be coloured red, while AllRed becomes green. All intermediate stages in the animation (in this case, just the transition's edge), become amber. Moreover, notice that the value of ELight in the current state is entered in the Values list, since its value (Green) is determined by the action of the transition.

The transition EWChangeGreen contains a predicate which should be true before the transition can fire: (LastGreen = NS); this predicate is added to the end (bottom) of the Path Condition box. By our choice of initial conditions, the predicate is not true, and indeed evaluates to False in the Path Condition box! Note that no such choice is made in the specification of the machine, and thus the state machine is modelling different choices of traffic light systems. This could be refined if desired.

Consistent with the choice of initial conditions, then, the user should choose the cycle starting with N/S lights changing. Select < from the set of buttons at the base of DOVE's main window; the animation will return to the previous state, with AllRed as the current state. Now proceed forward in the animation through the NSChangeGreen transition. Note that after this step the values of both Nlight and SLight are Green, as expected, and the Path Condition is True.

The WaitEW transition does not seem to "do" anything, but is required to ensure the machine cannot deadlock. Proceed through it and see the Path Condition which ensures this.

Now proceed through the remaining transitions in this cycle (including WaitTO), until the current state is again AllRed. After the last transition, NSChangeRed, the value of LastGreen is finally set, to NS.

The path condition which has built up is

```
(Not (MaxCars < (ECars_1 + WCars_1)))
(MaxCars < (ECars_2 + WCars_2))
(Not ((time_2 + MaxTime) < time_3))
((time_2 + MaxTime) < time_4)
```

Note that the subscripts signify that input values are completely independent from one step in the animation to another. Also, observe that the “wait transition” at a given node gives the complementary predicate to the precondition of the other possibile transition at that node. In this way, deadlock is avoided.

No name has been associated with the animation as yet - it is therefore known as an ”anonymous” animation, or the ”current” animation. If the user saves the state machine and exits DOVE at this point, this animation will be lost. However, the user may choose to save this animation, by selecting the option `Store` from the `File` menu of the `Animator` window. This produces a dialog which enables the user to enter a name for the animation, and a descriptive comment; note that the name must be unique for all the animations of the current state machine. The `Store` option can also be used to update the animation, should further changes be made to it. It is, however, important to be aware that the `Store` option does not do anything to the representation of the state machine in secondary storage. As with other state machine changes, the `Save` option of the `File` menu on DOVE’s main window must be applied if the animation is to be stored in a more permanent fashion. If this has not been done, the user will be prompted when exiting animation mode.

The user should now select the `New` option from the `Animator`’s `File` menu; this clears the animation, and any further animations which are created will again, be anonymous. To retrieve the animation that was previously stored, the user should choose `Select` from the `File` menu - this displays a submenu of animation names. By choosing the name of the animation that the user stored earlier, the graph should again be cleared of animation paths, and the stored animation, in its entirety, should be displayed.

The buttons at the base of the graph provide a convenient mechanism for stepping rapidly through an existing animation (although the user should be aware that if the animation steps are not present within the state machine theory, DOVE may have to replay the animation from scratch the first time it is loaded). By choosing the leftmost button marked |<<, the animation jumps back to the start, so that only its start state is coloured. Likewise, the rightmost button marked >>| will cause the animation to jump to the finish. The user can also apply buttons which cause the animation to proceed forwards or backwards for a given number of steps, or which replay either to the finish or the beginning of the animation. For a full discussion of these operations, please refer to Section 5.2.3.

To exit this animation session the user should click on either the edit mode or proof mode buttons, and answer `Yes` to the dialog box.

Chapter 6

Managing Proofs

DOVE provides powerful facilities for proposing, organising and tracking the status of state machine properties. The actual proof of properties is delegated to the XIsabelle graphical proof environment [2]. In this chapter the management of properties via the `Properties Manager` window is considered first, and then a tutorial section is provided to reinforce this material. After this, the `Prover` (XIsabelle) and `Theorem Browser` windows are discussed. The next chapter provides a discussion of DOVE's specialised proof strategies.

6.1 Window display on entering proof mode

Proof mode is entered by clicking on the "turnstile" button (see Figure3.2(c)) on the menu bar of the DOVE state machine window. Initially this brings up the `Properties Manager` window shown in Figure 6.1, and the `Prover` window shown in Figure 6.3.

Before the `Prover` actually appears, DOVE automatically calls the `Check/Compile` option (from the `Edit` menu of the state machine graph window). It is not possible to start a proof of a property in a state machine theory with fatal errors in the ensuing diagnostics. If no fatal errors are found, the corresponding diagnostic window will automatically disappear in a few seconds.

6.2 The `Properties Manager` window

The `Properties Manager` is accessible in Edit mode via the `Property Definitions` option of the `Definitions` menu, and is also brought up automatically when a proof session is initiated. It organises a database of properties, providing facilities for

1. maintaining named sets of properties;
2. defining and naming proposed properties within those sets;
3. editing existing properties;
4. entering existing properties into the `Prover`;

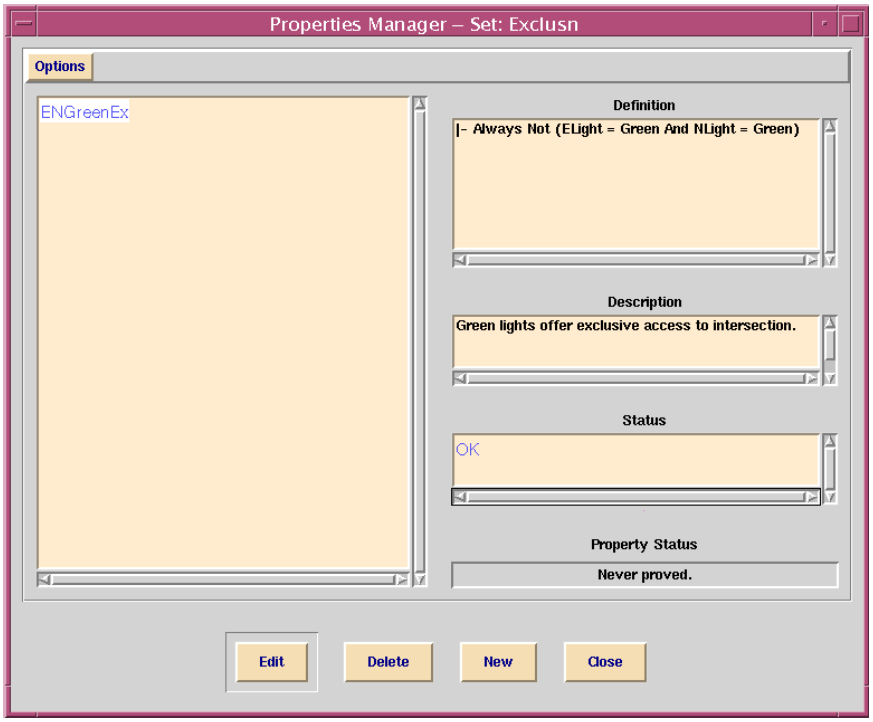


Figure 6.1: The Properties Manager window.

5. tracking the proof status of properties.

This functionality is implemented as a dialog box based on the declaration windows of the `Definitions` menu of the state machine graph window. The basic interaction – covering the second and third items of the functionality list above – is as described in Section 4.2.4, so only the distinguishing features will be discussed here. There is a general discussion on state machine properties in 2.7 and there is a discussion of how to go about formulating properties in Chapter 7. The exact syntax for properties is presented in Appendix B.

6.2.1 `Property Status` reporting

A new element of the declaration window is the additional area marked `Property Status`. As opposed to the `Status` area, which reports successful (or otherwise) parsing upon committal of a new entry, the `Property Status` area reports whether or not the selected property has been proved in the current state machine. Thus it implements the fifth item of the functionality list. There are three possible reports.

Never proved meaning that the user has not successfully proved the selected property at any stage of the development of the current state machine.

Not proved for the current state machine meaning that it has been proved in the past, but not with the current version of the state machine. In particular, this is the status if the state machine was saved since the selected property was proved. A time stamp indicating the time of committal of the most recent proof in a previous version of the state machine is given.

Proved for the current state machine meaning the user has proved the property in the current version of the state machine. A time stamp indicating the time of committal of the proof is given.

It should be noted that, after a property is proved in the Prover, the state machine must be saved so that the fact that the property was proved is recorded. If the time recorded for the saving of the properties file is older than that of the state machine graph then the Properties Manager will display the property as not being proved for the current machine. To save the state machine, simply use the Save option in the File menu of the state machine graph window, which is enabled when applicable.

6.2.2 The Options menu

The other distinguishing feature of the Properties Manager declaration window is the Options menu, whose contents implement the first and fourth items above, and are now described.

Set Manager starts up a dialogue box for selecting existing, or entering new, sets of property names. It can be convenient, say for logically ordering the state machine documentation, to have related properties gathered together. The Properties Manager allows this via the Set Manager window which is called up by selecting this option. The desired name for the set is typed directly into the Current Set text-entry box. Upon clicking on the OK button, the name is automatically inserted in the Available Sets list and the Set Manager window is closed. Existing names can be selected in the Available Sets list, and then renamed or deleted via the other buttons at the base of the Set Manager window. The selected name then becomes the current set.

The property names appearing in the declarations list of the Properties Manager window are those in the set whose name is printed in the title bar. This name changes to the current set as selected via the Set Manager window, and the properties of that set are then displayed. When first opened, the Properties Manager comes up with the current set automatically being called default. This can be modified as discussed above if desired.

Clear Properties removes all properties in the current set.

Save does nothing beyond the Save option under the File menu of the state machine graph window, and so is superfluous. However, its presence may help the user to remember to save the property, particularly after proof so that the Property Status is updated.

Prove Selected Property brings the Prover window to the foreground, initiating a proof session for the selected property. The Prover window is the XIsabelle interface. It is in this window that proof steps are carried out, and through this window the user interacts with the proof. A discussion of the Prover window is deferred to Section 6.4.

It is not possible to start up the Prover window if no property is selected. Also, note that invoking the Prover requires an up-to-date image file. DOVE will create one automatically if this is not the case.

If the Prover has already been started, selecting a new property and using the Prove Selected Property operation will cause that property to be loaded into the Prover as the Goal. User confirmation is required for this. The current proof session before loading can be re-accessed via the Swap menu on the Prover window, as discussed below.

6.2.3 Exiting the proof mode

To leave the proof mode the user must click on either the edit mode or animation mode button of the DOVE state machine window. A dialog box then appears, asking if the exit should continue (since the change of mode will result in an open proof session being discarded). If the user answers No then the current proof session is resumed. If the user answers Yes then the current proof session is cleared – *all* related open windows – and closed (and the desired mode opened).

If the user clicks on the proof mode button whilst currently in proof mode then a dialog box appears, asking if the proof session should be restarted. If the user answers No then the current proof session is resumed. If the user answers Yes then the current proof session is cleared, but the Properties Manager window remains open (since it is accessible via the Property Definitions option of the Definitions menu of the state machine graph window.

6.3 Tutorial: example properties in TrafficLights

In this tutorial the proof management process is illustrated by considering some example properties. To profit from it, the reader should have at least scanned Chapter 2; in particular, Section 2.1, which describes the various attributes and inputs of the traffic lights system.

It is assumed that the reader has the DOVE tool appropriately set up as in Chapter 3, and is in a working directory TrafficLights – which includes the file TrafficLights.smg – as discussed in the preamble to Section 3.2. In the following, the command-line prompt is denoted by the > at the beginning of the command line. Also, in common with the rest of the manual, grammatical notation will not be included in command lines.

Open a DOVE session with the TrafficLights state machine. As explained in Section 3.1, this is done by typing

```
> dove TrafficLights
```

The grey, grid-patterned canvas with the complete TrafficLights state machine will quite quickly appear on the screen. This is the DOVE state machine window, which was used for designing the machine – as discussed in Chapter 4, and in the tutorial Section 4.5. The user will notice that the "pencil" icon on the menu bar is blue, whereas the other icons are green. This makes manifest that the DOVE session is in editing mode. Click on the "turnstile" icon which will then be the only blue one, indicating that the session is now in proof mode. The Properties Manager window appears, and the Prover window follows.

The properties considered in this tutorial are the basic safety requirements of the TrafficLights machine; namely, that the E/W and N/S sets of lights are internally synchronised and that it is never the case that both sets of lights are green at once.

The synchronisation properties simply require that the variable pairs ELight and WLight, NLight and SLight, respectively, are always equal. Recalling the temporal operators presented in Section 2.7, this property may be expressed by the statement

```
|- Always (ELight = WLight)
```

in the case of the E/W lights, and by the statement

```
|- Always (NLight = SLight)
```

in the case of the N/S lights.

The exclusion property may be expressed by the statement

```
|- Always Not ((ELight = Green) and (NLight = Green))
```

It is not necessary to consider the other combinations of lights, provided that the synchronisation properties have been proved.

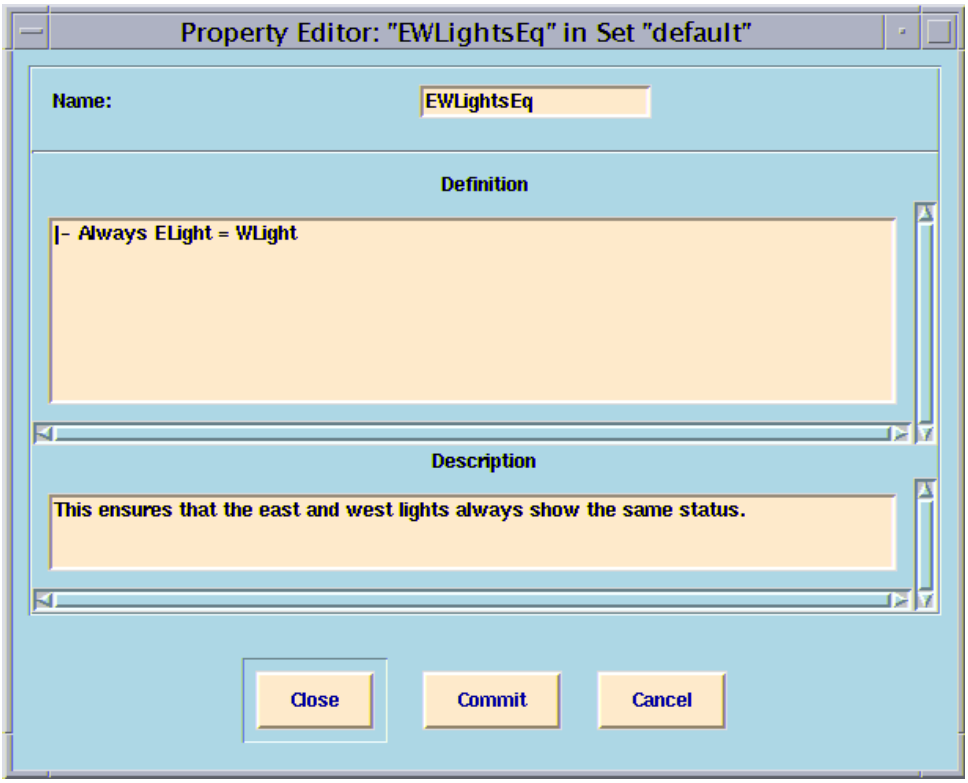


Figure 6.2: The edit property dialog window.

These properties will be entered in two different sets, labelled by the type of property as indicated above: synchronisation properties in the set “Synchronisn”, and exclusion properties in the set “Exclusn”.

The user should now create the set Synchro, and enter the synchronisation properties, as follows.

- Click on the Set Manager option of the Options menu on the Properties Manager window.

- Enter the set name Synchro in the Current Set: text-entry box of the dialog box which appears.

- Click OK on the dialog box. The Properties Manager window's title bar is now labelled by the new set name, and all properties entered here will belong to the set Synchro.

- Click on the New button on the Properties Manager window.

- Enter the name EWLightsEq in the dialog box which appears.

- In the Definition text entry field type the E/W synchronisation statement described above and in the Comment field type a descriptive sentence about the property. The edit dialog should now appear as presented in Figure 6.2.

- Click on the Commit button to ensure that the property is correctly entered.

- Click Close in the dialog box. The property EWLightsEq is then automatically inserted in the declarations list of the Properties Manager window.

- Similarly, enter the synchronisation property NSLightsEq as given above.

Similarly, create the set Exclusn and enter the exclusion property ENGreenEx. The Properties Manager window should now appear as presented in Figure 6.1.

A proof session for one of the properties can be initiated by selecting the Start Proof item from the Proof menu. In the next chapter the proof of the property EWLightsEq is considered in detail.

6.4 The Prover window

As stated in Section 6.2.2, the DOVE Prover environment is invoked by highlighting a property in the Proof Manager window and selecting the Prove Selected Property option from the Options menu. At this point the XIsabelle environment is brought to the foreground, as depicted in Figure 6.3.

The Prover display has been specialised in a number of ways to help support the proof of state machine properties. In order to be able to discuss these enhancements, a quick overview of the various features of the Prover window is given in the following subsections. The discussion here is worded so that it is not necessary to have a detailed understanding of the user/prover interaction induced from the proof model adopted by Isabelle (and hence XIsabelle). An analysis of the proof model itself appears in the next chapter. For any unfamiliar terms the reader should consult the glossary at the beginning of the User Manual. For more details of XIsabelle, the user should consult [2].

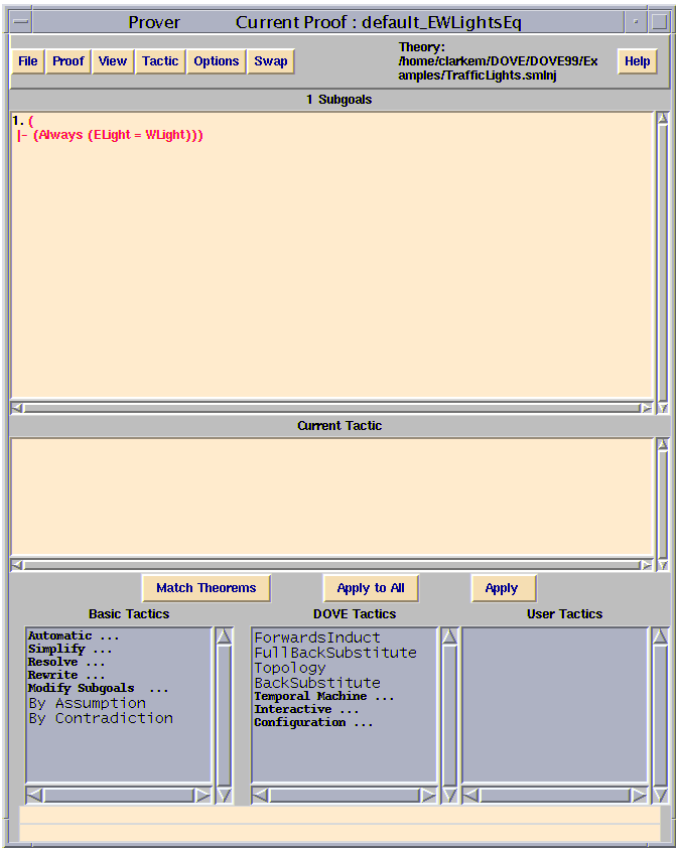


Figure 6.3: The prover window

6.4.1 Frames and buttons

Isabelle adopts what is termed a *goal-directed* proof strategy in which an evolving proof is represented as a stack of *proof states*. The list of currently unjustified results (called *subgoals*) provides the representation of the current proof state which the user sees. The user constructs the next level of the proof (the new proof state) by applying a valid proof step (a *tactic*) from the DOVE deductive system.

The `Subgoals` *frame* is the top frame of the `Prover` window, in which the proof state – the list of currently unproved subgoals – is displayed. The header of this frame actually displays the *number* of subgoals remaining. Initially, the subgoal list consists solely of the overall goal to be proved, but during the course of the proof various intermediate subgoals will be introduced. At each stage, the current subgoal to which a proof tactic is to be applied can be selected by clicking on it with mouse button 1, and is then highlighted by being displayed in red. The highlighted subgoal is called the *active subgoal*.

The `Tactics` *frames and buttons* are the frames, buttons and tactics lists below this top frame which allow the user to carry out proof steps. The basic procedure is that the user selects a proof tactic with mouse button 1. The chosen tactic is then displayed in the `Current Tactic` frame, and can be applied to the active subgoal by clicking on the `Apply` button. [The user may also simply double-click on the tactic.] Alternatively, it can be applied to all subgoals in the current proof state by clicking on the `Apply To All` button. After the proof tactic has been applied,

the top frame of the Prover window is updated to display all current subgoals (with the active subgoal highlighted). This basic procedure applies in particular to the list of DOVE Tactics, which contains proof tactics (discussed in detail in the next chapter) designed specifically for reasoning about DOVE state machines.

The use of Basic Tactics is similar, but requires slightly more knowledge of the XIsabelle proof tool. The user should refer at this point to the XIsabelle user manual [2] and the Isabelle documentation [9].

The User Tactics frame lists special tactics installed directly by the user. A brief application of this functionality will be given in the tutorial of the next chapter.

The Match Theorems button brings up the Theorem Browser window, toggled to Matching On, which lists the theorems that can be applied to the active subgoal. This facility is typically used in conjunction with many of the Basic Tactics, or with the Interactive sublist of DOVE Tactics, as discussed in the next chapter, and is a feature for the advanced user.

6.4.2 The menu bar

Across the top of the Prover window there is a menu bar with menus providing a variety of facilities: for customising XIsabelle's proof environment; for saving and managing proofs; for automating repetitive proof steps; and for printing proofs in human-readable format. To find detailed descriptions of the functions provided by these menus the reader should refer to the XIsabelle user manual [2], which can be obtained under the Help menu button. The present document is *not* self-contained in that these details are typically not repeated here. However, it is worthwhile emphasising the menu items which are particularly relevant for the DOVE tool, and the following lists some of the relevant highlights.

6.4.2.1 The File menu

is largely self-explanatory, providing the functionality for proof management – in particular for saving proofs into proof scripts. More discussion of this proof management will be given below in Section 6.5. The option Export as LaTex produces a user-readable account of a given proof.

6.4.2.2 The Proof menu

includes Chop facilities, which are used to undo proof steps; either the Chop option which undoes one proof step, or the Chop to Level option which chops back to the specified level (that can be read from the current Proof History), or the Restart option which chops back to the start. The Fast Forward option can then be used to scroll quickly back through all proof steps to the final level in the current Proof History, while the Step Forward option applies just one step. The keyboard accelerators for these options are:

- Control-left for Chop
- Control-right for Step Forward

- Shift-left for Restart
- Shift-right for Fast Forward

6.4.2.3 The View menu

provides different modes of proof visualization in the Tree Display and Proof History options. The Proof History options bring up a *proof script* which lists, in order, the proof steps applied before the current proof state (or the complete list in a previously-completed proof). These options should be accessed during proofs to provide simple means of applying proof steps – simply double-clicking on the required step. Another option in this menu, the View Theorems option, brings up the Theorem Browser window (toggled to Matching Off by default, or to the previous state in which the Theorem Browser was quit). See Section 6.6 for more details on the Theorem Browser window.

6.4.2.4 The Options menu

provides a Subgoal Display option that has a subset of the options in the Options menu of the Theorem Browser window, which is discussed in Section 6.6.

6.4.2.5 The Swap menu

provides the facility to swap between proof sessions for different properties which have been loaded concurrently from the Properties Manager, as discussed later in Section 6.5.

6.5 Proof management: ending, saving, loading and restarting proofs

In this section several useful hints for managing proofs are collected.

- The proof of more than one property can be attempted concurrently – which is useful, for example, if the user realises that a further lemma is required – as will now be explained. At any stage in a current proof session the user may enter a new property into the Prover via the Properties Manager window, following the steps explained in Section 6.2.2. This starts a new proof session for the new property (user confirmation is required). To change the Prover window to display the appropriate proof session, the user may simply choose the desired option in the Swap menu. In this way, a total proof session builds up, consisting of any number of concurrent component proof sessions. In particular, a completed proof will remain as a component of the total proof session.
- Once the total proof session is exited – as explained in Section 6.2.3 – the proof steps to date in all proofs are lost unless the proof script has been saved (up to its current state) by the user. This is achieved by selecting the Save option of the File menu on the Prover

window. The proof script containing all steps to the current proof state for a given property P, say, is then saved to a file P.prf. This proof script can be re-exhibited by choosing the Proof History (Other) option of the View menu on the Prover window, which then brings up a dialog box from which the user may select the desired script file. Upon selecting it, a Proof History window appears which contains the script.

- The proof scripts are saved in XIsabelle format, so that a given proof step may be replayed – or even inserted into the Prover while in a completely different proof session – by clicking on the corresponding line in the script and then pressing the Apply button (or, alternatively, by simply double-clicking on the corresponding line in the script).
- Because of this tactic insertion facility, it is convenient *always* to have open a Proof History window for the current proof, which is achieved by choosing the Proof History (Current) option of the View menu on the Prover window.
- On the completion of a proof, DOVE will announce to the user that the proof is complete and then wait for the user to click OK. DOVE then automatically adds the corresponding theorem to the theorem list of the current theory, and commits a new ML image of the updated theory (this may take a little time). The theorem will immediately appear in the Theorem Browser window (in the Theorem Names list corresponding to the name of the current state machine in the Theory Names list).
- The proof script is *not* saved automatically at the completion of a proof. It is good practice to save the proof script for later consideration. Saving is carried out as discussed above. A different name for the proof script may be given by instead using the Save As option of the File menu.
- When quitting the Prover while it contains an unsaved proof, DOVE will display a dialog box asking the user if the proof should be saved or not.
- Since unfinished proofs can be saved, the user may quit DOVE and resume the proof session at a later date simply by:
  - entering the proof session and loading the desired property into the Prover through the Properties Manager as usual;
  - bringing up the desired proof script via the Proof History (Other) option of the View menu on the Prover window, as discussed above;
  - re-entering the proof steps known to date by double clicking on the corresponding line in the proof script.
  - When quitting the DOVE session completely, if any proofs have been completed then a new ML image of the current logic state will automatically be written. Thus, when a later DOVE session with the same state machine is commenced, these proved properties will be available as theorems for use in proving further properties – without needing to be reproved.
- As discussed above, the Save option of the File menu on the Prover window saves the proof scripts in XIsabelle format. To obtain a more human-readable form, use the Export as LaTex option of the File menu on the Prover window. This will replay the proof, and generate a LaTeX file (in the directory where the DOVE session is being run) containing the proof.

6.6 The Theorem Browser window

The Theorem Browser is adjunct to the Prover window. It simply contains a list of the theorems which are available for the user to reason with whilst carrying out the proof in the Prover window. This is a feature for advanced users.

The Theorem Browser window can be brought up via the Prover window via the View menu, or via the Match Theorems button, as discussed above in Section 6.4.1. The Theorem Browser window will come to the foreground when invoked, and may be quit independently of the Prover via its own File menu option, Close.

6.6.1 Frames

There are three frames on the Theorem Browser window, which all have a very different functionality.

The Theory Names frame lists the names of all Isabelle theories upon which the current state machine is based (including the current state machine!). The name of the currently selected theory is recorded on the RHS of the menu bar.

The Theorem Names frame lists the names of theorems which are contained in the selected theory. In particular, any theorem proved by the user in the DOVE session will be included in this list. The actual class of theorem names which will be displayed here depends on the menu options chosen, as discussed below. This frame is "clickable", and is used to insert the theorem into the Prover:

- single-clicking on a theorem name with mouse button 1 will display the statement of the theorem in the Theorem frame.
- holding down the Shift key and single-clicking on a theorem name with mouse button 1 will apply the theorem to the active subgoal in the Prover window (if possible) via an automatically chosen tactic.
- clicking once on a theorem name with mouse button 2 will insert that name into a waiting tactic in the Current Tactic frame of the Prover window. As usual, the tactic can be then be applied to the active subgoal by clicking on the Apply button, or to all subgoals in the current proof state by clicking on the Apply To All button.

This usage will become clearer after the user has gone through the tutorial in Section7.5.

The Theorem frame displays the theorem corresponding to the theorem name chosen by single-clicking with mouse button 1 in the Theorem Names frame.

6.6.2 The menu bar

The File menu provides an option Close for closing the Theorem Browser window.

The View menu controls which class of theorems is displayed in the Names frame. The typical user would simply choose the All Theorems option. The advanced user may want to

restrict the output appropriately. The Temporal Facts option is used in conjunction with the Add Temporal Fact option in the DOVE menu of the Prover window, as described in Subsection 7.3.3.

The Options menu provides an option Display which allows various different displays of the subgoals. Although the typical user is unlikely to make use of this menu, the toggles for Show Brackets and Show Names in the Display option are DOVE additions to XIsabelle. The option Show Brackets enforces the display of the brackets which make manifest the bindings of various operators in the subgoal terms. If the Matching On facility is required then it is crucial for this option to be toggled on, since DOVE requires the brackets to be able to correctly parse the terms. The option Show Names, if toggled on, expands the abbreviations in the DOVE syntax for temporal operators, thus making them more readable.

The Help menu contains different options for XIsabelle and Isabelle documentation. These may have more information about the functions on the Prover and Theorem Browser windows beyond that given in the present DOVE User Manual, if required.

6.6.3 Matching terms facility

When the Theorem Browser window is brought up from the Prover via the Match Theorems button, the tool will match on the form of the active subgoal in the Prover window. Thereby, the Theorem Browser displays only those theorems which will have an effect when applied appropriately to the active subgoal.

6.7 Proof visualization

One approach to building better formal methods tools is via the concept of *proof visualisation*: a combination of techniques, built on top of existing automated proof tools, that helps the user understand, structure and control long and complex sequences of reasoning in a natural way. These techniques involve the use of diagrams and pictures to represent aspects of formal proofs, but also include the use of domain knowledge in proofs and the construction of appropriately-sized proof steps.

One effective way of structuring complex information and processes is by means of graphical representations, or *visualisation*. There are two ways in which such a representation can be used:

- *passive*, which helps the user *understand* and *communicate* the proof; and
- *active*, which allows the user to *control* construction of the proof.

Passive visualisation has been implemented in DOVE, which highlights the node and edge corresponding to the state and transitions occurring in the assumptions of the current active subgoal. This allows one to see which part of the design is currently being reasoned about.

The active subgoal can be changed by clicking on a transition entering a state node, thus demonstrating a very basic form of active proof visualisation. This triggers a search through the subgoal list for the first one whose hypotheses match the chosen transition and state (which are, in

turn, highlighted). The matched subgoal becomes the new active subgoal. Selection of subgoals from the graph in this way can conveniently allow the proof to be developed in an order that appear natural from the design, rather than the more arbitrary order in which the subgoals are listed. The reader may wish to experiment with this facility during the proof tutorials in Chapter7.

Chapter 7

Proof strategies and tactics

It has been found that the formal proof facility is the aspect of the DOVE tool which is hardest for beginning users to grasp. This is not surprising in itself, since the rather baroque intricacies of syntax, semantics, and logic in formal methods tend to test the attention-span of even the most dedicated mathematicians. However, some effort has been made in the DOVE tool to eliminate much of the arcane notation, conventions, and artifices which are required in the construction of a formal proof – at least, as much as is practicable without losing the ability to make nontrivial applications. In particular, a proof strategy suitable for a wide range of DOVE state machines has been devised and encoded as a set of steps to be implemented in the `Prover`.

To explain this, and yet to cater for more experienced users, the present Chapter is written at two different levels. The introductory level is relatively straightforward, with the basic DOVE proof strategy presented as a recipe and demonstrated on a simple example. The advanced level presents more details and has a discussion of more complicated tactics demonstrated in a somewhat harder example. Specifically, the split is as follows.

- The introductory level, building on the outline of good `Prover` practice given in Section 6.5, and the earlier Sections 2.7 and 2.8, is found in:
  - the glossary on page (xv) of this User Manual, which gently introduces the language of formal methods;
  - Appendix G, which introduces the concepts of formal theories.
  - Section 7.1, which gives an explanation of the nature and workings of the proof assistant Isabelle, and a discussion of the DOVE implementation;
  - Section 7.2, where the DOVE proof tactics are introduced – particularly Subsection 7.2.1, where the fundamental tactics of the DOVE proof strategy are given; and
  - Section 7.4, a tutorial demonstrating those fundamental tactics on a simple example of an invariant property of the `TrafficLights` machine.
- The advanced level is found, beyond those sections, in:
  - Appendix C, where a technical presentation of the inference rules provided by DOVE, one more suited to the expert reader, can be found;
  - Section 7.5, a tutorial demonstrating more advanced use of DOVE tactics.

– Section 7.6, which has further discussion of the use of the proof tool, concentrating on the methods used in the advanced tutorial.

In this way it is hoped to provide an intuition for the basics of proof in DOVE and to demonstrate the level of power available even to the naive user.

The verification of state machine properties in DOVE is carried out using the graphical proof environment XIsabelle [2]. This manual does not attempt to provide a detailed introduction to the XIsabelle tool. Instead, the unique aspects of proof in the DOVE environment are emphasised – in particular, the specialised strategies, tactics, and inference rules provided specifically for reasoning about state machine properties. However, to implement the *full* power of the `Prover` it is necessary for the user to be familiar with the Isabelle proof tool. This is, perhaps surprisingly, not a particularly onerous task, as the interested reader will find on studying the existing user documentation for Isabelle [9] and XIsabelle [2].

7.1 Proof in DOVE/XIsabelle

Before presenting the DOVE proof strategy, some of the basics of how Isabelle mechanises the process of logical reasoning are considered.

7.1.1 Interactive proof tools

Isabelle is an *interactive* proof tool, meaning that it implements proof tactics which the user must select at each stage of the evolving proof. A common question is: since many steps of a formal proof are quite trivial, why not use an *automatic* proof tool which simply takes the goal and attempts to construct a formal proof? There are a number of important reasons why the interactive tool is better suited to the task here.

- It is not possible to give a general algorithm which will construct a formal proof for any property. Thus an automatic tool needs to be tightly targeted to a subclass of subgoals. A specific and fixed logical structure is introduced to carry this out. In an interactive proof tool such as Isabelle the logical structure can be extended almost indefinitely, which clearly provides the user with a lot of power. In DOVE the aim is to harness this power, while directing it in a more accessible manner.

- It is *not* necessarily a good idea to have tactics which make a large change from one proof state to the next, since the user often needs to be able to keep track of the logical changes involved. Automatic proof tools clearly provide an extreme example where the final proof state is very far from the original goal. The machine-generated proof is unlikely to be very intuitive or useful for the user. Moreover, when the automatic proof attempt does not succeed there will be essentially no information to be gained by looking at the current proof state. The user must then devise clever lemmas which the tool can address successfully on its way towards proving the overall goal. The need to develop this skill is a huge burden on the user.

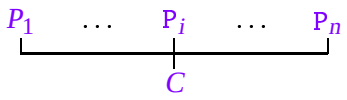


Figure 7.1: The structure of a theorem (or proof state).

- A major use of verification is directed towards the certification of products to high levels of trust as embodied in, say, the ITSEC [3] classification. For such situations the level of trust must be transferred from the verifiers to the evaluators, so a well laid out sequence of small and easily checked proof steps is optimum in that setting.
- An interactive proof still requires that the small steps be structured towards proving the goal. However, this structuring in itself provides the user with deeper insight into the state machine properties. Failure of the proof attempt can then possibly be correlated with a defect in the machine design.
- At the same time, some convenient level of automation can be implemented in Isabelle by bundling together individual inference rules to make specific tactics. In DOVE such tactics are available in the `Prover`, often allowing the optimum proof strategy to be implemented in just a few steps.

7.1.2 Theorems and inference

At the core of Isabelle is an abstract datatype for representing the terms and logical propositions to be reasoned about. Associated with this datatype are a small number of trusted functions for constructing logical propositions which are guaranteed to be "true". These functions are called the *inference rules* of Isabelle and the true propositions they construct are called *theorems*.

Theorems in Isabelle can take either of two forms:

- A logical implication,

$$\frac{P_1 \ldots P_n}{C}$$

  states that a *conclusion* C is true whenever the *premises* P_1, ..., P_n are true. Figure 7.1 presents a useful graphical depiction of a logical implication.

- An equivalence theorem,

$$S \equiv T,$$

  states that the term S has exactly the same meaning as the term T.

Three of the most important inference rules in Isabelle are assumption, resolution, and substitution. The assumption rule (see Figure 7.2) builds, for any proposition P, the theorem

Figure 7.2: The assumption inference rule

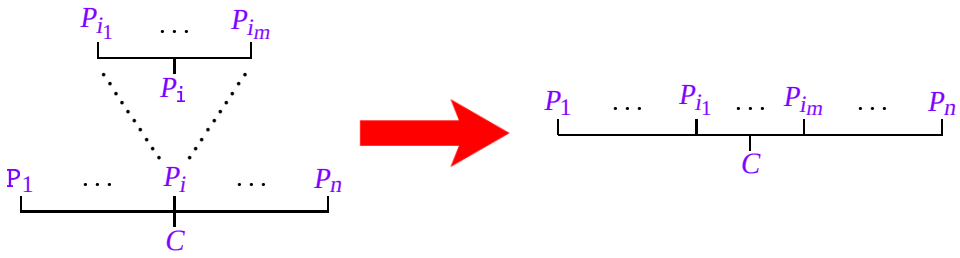


Figure 7.3: The resolution inference rule

$$\frac{P}{P}$$

The resolution rule (see Figure 7.3) joins an implication,

$$\frac{P_{i_1} \ldots P_{i_m}}{P_i}$$

with an implication with premise P_i,

$$\frac{P_1 \ldots P_i \ldots P_n}{C}$$

by replacing the premise $P_i$in the latter implication with the premises $P_{i_1} \ldots P_{i_m}$of the former. The substitution rule uses an equivalence $S \equiv T$ and a theorem $P(S)$ in which the term S occurs, to construct the new theorem $P(T)$, in which S is replaced by T.

Inference rules such as those described above are called *pure* rules because they construct theorems without regard to the meaning (or semantics) of the terms which they manipulate. Obviously, if Isabelle did not allow one to make use of the meanings of terms, it would not be a very useful tool. The mechanism Isabelle uses to make use of the meaning of terms is the *theory* file. A theory file introduces a collection of named objects and a collection of theorems about those objects, called the *axioms* of the theory. For example, the file `TrafficLights.thy` is the theory file generated by DOVE to tell Isabelle the meaning of the `TrafficLights` state machine. It introduces all the states, transitions, variables, etc. of the `TrafficLights` state machine and a collection of axioms that can be used to reason about it.

7.1.3 Proof-state and tactics

In order to facilitate interactive proof, Isabelle provides what is termed a *goal-directed* proof package. The aim of the package is to allow the user to reason *backwards* from a desired property (called the *goal*), through intermediate premises (called *subgoals*) which are sufficient to establish the goal, finally to the axioms (or previously proven theorems) of the theory. The XIsabelle tool provides a graphical environment for performing such goal-directed proofs. In the following, only proof in XIsabelle is considered.

The first step in an XIsabelle proof is to enter a desired goal `G` and the theory in which it is to be proved[11]. XIsabelle then uses the assumption rule to construct the theorem

$$\frac{G}{G}$$

which becomes the initial *proof-state*. At each stage in the proof, the proof-state will consist of a theorem of the form

$$\frac{SG_1 \ldots\ SG_n}{G}$$

(together with added bookkeeping information used to support the mechanics of the XIsabelle proof activity). The premises of the proof-state are called the current subgoals and it is these that are displayed in the XIsabelle prover window (see Figure 6.3).

In order to progress the proof, the user applies a *tactic*, which, simply put, is just a program which changes the proof-state. Most tactics work by making use of inference rules, axioms, and known theorems to replace one of the subgoals with zero or more new subgoals. Thus the most common form of proof step in XIsabelle consists of selecting a subgoal (from the main prover panel), selecting a tactic (from either of the `Basic Tactics` or `DOVE Tactics` panels), and then possibly selecting one or more axioms or theorems (from the `Theorem Browser` window) for use in the tactic. If a tactic replaces a subgoal by an empty collection of new subgoals, the subgoal is said to have been *discharged*. The aim of the proof activity is to discharge all of the subgoals, leaving a proof state of the form

$$\frac{}{G}$$

which is just the statement that the property G is a theorem.

As already stated, XIsabelle keeps a certain amount of book-keeping information in the proof-state. An important part of this is the *proof-history*, which records faithfully all of the tactics that have been applied in developing the current proof-state. This can be used to backstep to an earlier stage in the proof or to repeat a certain sequence of steps from earlier in the proof. It is even possible to open proof-histories from other proofs entirely and make use of them in the current proof.

[11] In DOVE this is handled by the `Properties Manager`.

7.1.4 Temporal sequents

Although any form of logical subgoal may appear in a DOVE proof, subgoals generally appear in the form of a *temporal sequent*. A simple form of the temporal sequent has already been seen in Section 2.8. Each sequent consists of a list of temporal *hypothesis* formulae and a temporal *target* formula, separated by a sequent turnstile.

H_1, ..., H_n |- T

In order to prove such a sequent subgoal, it is necessary to demonstrate that the target is true, provided that all of the hypotheses are also true.

Appendix C explains in detail how the DOVE temporal logic theory is developed through the use of sequents. For the remainder of this chapter it is sufficient that the reader is familiar with the format of sequents, and with the terms hypothesis and target introduced above.

The XIsabelle tool (and the underlying Isabelle tool) provide a large number of powerful features not described in this section. The aim of this section has been just to introduce the vocabulary and concepts which are vital to an understanding of the proof examples which follow. The interested reader is strongly urged to consult the XIsabelle [2] and Isabelle [8] manuals in order to gain a better appreciation of the power these tools provide.

7.2 The DOVE proof strategy

As described in Section 2.8, the proof strategy adopted in DOVE is that of *induction* on the executions of the state machine, implemented via the process of *back-substitution*. Briefly, this works as follows.

1. Given a property, the question is whether it holds for all executions of the state machine. This can be answered in the affirmative, under the principle of induction, if:

   - all the initial configurations satisfy the property; and
   - for every execution of the state machine which satisfies the property, all enabled transitions take the state machine to an execution which also satisfies the property.

   The latter item is called the *inductive step*. In DOVE, the inductive step is reinterpreted as a statement in temporal logic.

2. Given a subgoal, a DOVE state machine will only have a finite number of nodes and transition edges, possibly all, which are relevant to the proof of that property. The subgoal can then be split into cases uniquely determined by this *topology* information from the graph.

3. Given a temporal property, one can clearly describe what must have been true in the *previous* configuration (defined uniquely after the decomposition under graph topology) in order for the current configuration in the execution history to satisfy it. DOVE *back-substitutes* to replace occurences of variable names in the corresponding temporal sequent with the values assigned to them by the last transition.

Once the back-substitution of a required property has been determined, the induction step becomes a ‘simple’ matter of proving that the required property implies its own back-substitution – or, possibly, that this is true after a number of back-substitution steps.

Items 1, 2 and 3 enumerated above form what will be called the *basic DOVE proof strategy* – or the “back-substitution proof strategy”. DOVE provides corresponding tactics corresponding to each step – respectively, `ForwardsInduct`, `Topology`, and `BackSubstitute` – in the `DOVE Tactics` frame on the `Prover` window. These tactics are discussed in more detail in Subsection 7.2.1 below.

More generally, in the `DOVE Tactics` frame DOVE presents the user with a small, but powerful, collection of “tailor-made” tactics that have been specifically designed to augment or support the basic back-substitution proof strategy. These tactics are automatically installed in the XIsabelle `Prover` when a new proof begins. They are discussed further in Subsections 7.2.2 and Section 7.3 below.

XIsabelle further provides facility for the user to install customised user tactics, as described in the XIsabelle manual [2], but in this section only the tactics supplied with the DOVE tool are considered.

The essence of the philosophy used in constructing DOVE’s verification tool is that, when augmented by some simple structural rules, the “temporal operations” in state machine properties are most effectively dealt with via the back-substitution strategy. Thus, while maintaining the power of the underlying Isabelle proof tool as much as possible, the `Prover` is nevertheless streamlined towards specialised use. The advanced user could simply take the `Prover` as an augmented front end to the Isabelle tool, but for design verification the `DOVE Tactics` will largely suffice.

7.2.1 The primary tactics of the DOVE proof strategy

To begin the discussion of DOVE’s user-tactics, consider the three tactics which implement the back-substitution strategy.

`ForwardsInduct` The `ForwardsInduct` tactic is likely to be the first tactic used in most DOVE proofs. It may be applied to any goal of the form

> `Always` A_1, ..., `Always` A_n,
> `Initially` I_1, ..., `Initially` I_m,
> H_1, ..., H_l
> `|- Always` P

and introduces the subgoal

> `Always` A_1, ..., `Always` A_n,
> `Initially` I_1, ..., `Initially` I_m,
> H_1, ..., H_l,
> `Previously Always` P
> `|- Always` P

This is simply the inductive hypothesis discussed earlier, as becomes clear if one was to decompose under `First` and `Not First` to get the equivalent representation in terms of two subgoals: the initial case

`First`, `Always` A_1, ..., `Always` A_n, I_1, ..., I_m |- P

(since any Previously hypothesis is trivial when `First` holds as explained in2.7), and the inductive step

`Not First`, `Always` A_1, ..., `Always` A_n, `Previously Always` P |- P

This splitting is not done explicitly by the `ForwardsInduct` tactic, since it is part of the `Topology` tactic decomposition.

`Topology` The `Topology` tactic uses the structure of the state machine graph to introduce the state and transition information necessary for applying the `BackSubstitute` tactic described below. It analyses the topology information already present in a subgoal's hypotheses to determine all the positions in the graph which are consistent with that information. For example, consider the subgoal

`At AllRed` |- P

in the case of the `TrafficLights` machine. Referring to Figure 2.1, it is clear that when in the `AllRed` state the machine is either in the first configuration, or else it has just performed `NSChangeRed` from the `NSAmber` state, or else it has just performed `EWChangeRed` from the `EWAmber` state. Thus, the `Topology` tactic introduces the following three new subgoals.

```
Not First,
Previously At EWAmber,
At AllRed,
By EWChangeRed
|- P
```

```
Not First,
Previously At NSAmber,
At AllRed,
By NSChangeRed
|- P
```

```
First,
At AllRed,
At AllRed
|- P
```

Thus,

`PreviouslyAt B`, `By T`, `At E`

uniquely identify the edge in the state machine graph with begin state node B, transition link T, and end state node E; i.e., Topology simply performs an edge decomposition consistent with the hypotheses of the given subgoal. Note in particular that Topology first decomposes under First/Not First. As a result, the Previously operator is always accompanied by Not First in the decomposed subgoal, which is formally nicer in that it always is manifestly well-defined on the configuration. This will be a feature of all further steps after the Topology tactic has been applied.

BackSubstitute The BackSubstitute tactic makes use of the definitions of the various transitions to determine what must have been true in the previous configuration in order for a subgoal to be true in the current configuration. To use BackSubstitute it is necessary to have hypotheses which precisely determine an exact edge in the state diagram corresponding to a particular transition. This information is generally introduced through the use of the Topology tactic and has two possible forms.

To use the definition of the initial condition, the subgoal must (in the case of the TrafficLights machine) be of the form

```
First,
At AllRed,
H_1, ..., H_l
|- P
```

To use the definition of any given transition, the subgoal must be of the form

```
Not First,
Previously At B,
At E,
By T,
H_1, ..., H_l
|- P
```

As discussed in Section 2.8, the back-substitution process replaces all occurrences of variables changed by the transition with their final values. In many cases the resulting subgoal is logically trivial and the BackSubstitute tactic is able to prove it without the help of the user. For example, consider applying back-substitution to the subgoal

```
Not First,
Previously At EWAmber,
By EWChangeRed,
At AllRed
|- ELight = WLight
```

Since the EWChangeRed transition has action

```
Act: ELight ← Red
     WLight ← Red
     LastGreen ← EW
```

back-substitution results in the new subgoal

```
At EWAmber |- Red = Red
```

which the BackSubstitute tactic immediately recognises as trivially true, and thus proves automatically.

7.2.2 Augmenting the basic strategy

In many cases, the initial property to be proved turns out not to be strong enough to be an invariant under the state machine's transitions. If this is the case the user will come to a situation in which the available assumptions are not adequate to prove the property, even with the use of the BackSubstitute tactic. In this circumstance there are two possibilities. Either the property is not true of the state machine or else it is true of the state machine, but is not invariant under the transitions.

The case that the property is not true may be recognised by the presence of a subgoal which includes the hypothesis First and which is false. The other case may be recognised by the presence of a subgoal including the hypothesis Not First, such that the subgoal cannot be proved even though it appears to be true of the state machine. In this second case the user should use Add Invariant to enter a new hypothesis which will help prove the required subgoal.

Add Invariant The Add Invariant user tactic is accessed from the Interactive submenu of the DOVE Tactics list (by *double-clicking* with mouse button 1).

For example, the subgoal

```
Not First,
At NSAmber Implies ELight = Red,
At NSGreen
|- ELight = Red
```

is not immediately proved by simply using back-substitution. However, it is clear that ELight is in fact Red in the NSGreen state. In order to proceed with the proof, the user may use the Add Invariant tactic to insert the additional hypothesis:

```
At NSGreen Implies ELight = Red
```

This converts the above subgoal to:

```
Not First,
At NSAmber Implies ELight = Red,
Previously Always (At NSGreen Implies ELight = Red),
At NSGreen
|- ELight = Red
```

which is straightforward to prove. Of course this benefit does not come for free, it is now necessary to prove the truth of the new invariant. DOVE also generates a new subgoal of the form:

```
Always (At NSAmber Implies ELight = Red)
|- Always (At NSGreen Implies ELight = Red)
```

which may be addressed using ForwardsInduct and the general back substitution approach. Precisely such an example is discussed in detail in Section 7.5.

7.3 The DOVE Tactics frame

The primary tactics of the DOVE proof strategy are accessed through the DOVE Tactics frame on the Prover. One more tactic appears in the primary list (MasterBlast). The remaining tactics appear when the bold-face submenu title is clicked on. Many subgoals which appear in the course of a proof are of a logical complexity such that more specialised tools are required to discharge them. This is the role of the remaining tactics (some of which are incorporated in MasterBlast), whose functionality is outlined below.

7.3.1 Primary tactics

The first three tactics in the DOVE Tactics list, Topology, ForwardsInduct, and BackSubstitute, have been explained in detail in Section 7.2.1. The fourth, MasterBlast, puts together the Topology decomposition with repeated application of BackSubstitute, BooleanSimplification and SolveTemporal (discussed below) to try to discharge the ensuing subgoals. Thus, it is applied at the point of the DOVE proof strategy where Topology would be applied (after the application of ForwardsInduct – actually, it is usually best to first apply DecomposeSequent (discussed below) to clean up the subgoal). In general it will then take the proof far from the stage at which MasterBlast was applied, well into the DOVE proof strategy.

The subgoals remaining after application of MasterBlast should either be solved by its further application, or will require implementing the Add Invariant tactic as discussed above in SubSection 7.2.2. In particular, it will directly prove most “simple” properties which largely depend on the graph topology for their veracity. Thus it is extremely powerful for the user who does not want a step-by-step record of the proof, and prefers the bigger jumps to points in the proof where additional invariants are needed.

7.3.2 Temporal Machine tactics

DecomposeSequent repeatedly applies some simple logical rules to eliminate disjunctions and conjunctions in the hypotheses and implications in the target. It is a good idea to apply this tactic regularly, so as to keep the subgoals small.

Contradictory Hypotheses solves subgoals which have hypotheses of the form P, Not P. Such subgoals are trivially true (“False implies anything”).

`BooleanSimplification` evaluates the truth values of simple sub-properties, such as

`Red = Green`,

and then simplifies complex hypotheses and targets using Boolean truth tables. For example, the hypothesis

`Green = Green Implies At NSGreen`

is simplified to

`At NSGreen`

since `Green = Green` is trivially true.

`UnwindLets` unwinds local definitions (cf. Section 4.3.1).

`SolveTemporal` is a more sophisticated version of `DecomposeSequent`, which combines unwinding of the derived temporal operators and the repeated application of the temporal decomposition rules. Unlike `DecomposeSequent`, it will fail and leave the proof state unchanged if it cannot discharge the subgoal.

`OpenAlwaysHyps` deals with temporal sequents with a hypothesis of the form `Always` *H*, where the facts inside *H* are needed to solve the subgoal. It "opens" the "Always" hypothesis to expose *H* and discharges the subgoal if possible via `SolveTemporal`-type tactics.

7.3.3 `Interactive` tactics

Note that to select the `Interactive` tactics appropriately the user must *double-click* (with mouse button 1).

`AddInvariant` was discussed above in SubSection 7.2.2.

`AddTemporalFact` requires the user to insert the name of the desired temporal property into the `Current Tactic` frame. The user then simply presses `Apply`, the result of which is to add the fact corresponding to the chosen temporal property to the hypotheses list of the current subgoal. The name can be inserted after placing the text insertion cursor in the `Current Tactic` frame. Alternatively, as directed at the bottom of the `Prover` window, bring up the `Theorem Browser` by pressing the `Match Theorems` button. Ensure that the current state machine theory is selected in the `Theory Names` list. Select the `Temporal Facts` option of the `View` menu on the `Theorem Browser`, and then select the theorem to be included from the `Theorem Names` list using mouse button 2. This will insert the theorem name into the `Current Tactic` window, and the user must then simply press `Apply`.

`ApplyTemporalRule` is a tactic for the advanced user. It resolves one of temporal rules – which typically reside in the MachCor theory – against the active subgoal, and deals with the technical subgoals which result. Otherwise it is similar to `Resolve (Intro)` in the `Basic Tactics` list. The name of the rule can be inserted into the tactic in precisely the same way as for `AddTemporalFact`.

`ApplyInstTemporalRule` is another tactic for the advanced user. It is an extension of `ApplyTemporalRule` which allows meta-variables in the rule to be instantiated. Thus, it is similar to `Resolve Instantiated (Intro)` in the `Basic Tactics` list (Isabelle's "res-inst-tac"). This is particularly useful when a given rule may unify against the active subgoal in a number of ways, so the instantiation is used to disambiguate the application.

7.3.4 `Configuration` tactics

`GetConfig` extracts simple configuration properties from the hypotheses and the target to produce a non-temporal subgoal. The `SolveConfig` and the `SimplifyConfig` tactics may then be used to discharge the subgoal. If these fail it will be necessary to reason using Isabelle's basic logical proof rules.

`SimplifyConfig` applies Isabelle's simplication package to a configuration (non-temporal) subgoal. Such sub-goals are normally introduced by the `GetConfig` tactic. This tactic will often discharge simple subgoals altogether.

`SolveConfig` invokes Isabelle's sophisticated classical reasoning package to solve a configuration (non-temporal) subgoal. Such sub-goals are normally introduced by the `GetConfig` tactic. If `SolveConfig` fails to discharge a subgoal the proof state is left unchanged.

Note that these configuration tactics take care of the other form of "contradiction" similar to `Contradictory Hypotheses`, but where the hypotheses are of the form `v = x`, `v = y` where `v` is a state variable and `x`, `y` are distinct constants. Simply reduce to a configuration property via `GetConfig` and then apply `SolveConfig` to solve by contradiction in the hypotheses.

7.4 Introductory Tutorial: the DOVE proof strategy

Recall from Section 2.8 that DOVE's recommended proof strategy makes use of induction and back-substitution. In this section this strategy is described in detail by applying it to the `EWLightsEq` property described in Section 6.3. This application uses the most important of the powerful proof tactics provided by the DOVE system. While applying the proof steps, the user should also watch the state machine graph to see the implementation of proof visualization as discussed in the last section.

It is assumed at this point that the reader will be familiar with the DOVE tool, at least at the level of the tutorials in Sections 4.5, 5.4, and 6.3. Hence, there will not be the same attention to pedagogy as in those sections.

The first step in proving the EWLightsEq property is to enter proof mode and highlight the property by selecting it in the Properties Manager window with mouse button 1. Then enter it into the Prover following the steps explained in Section 6.2.2.

7.4.1 Induction

The first step in a DOVE proof is generally performed by applying the ForwardsInduct tactic. This may only be applied to a subgoal of the form

H_1, ..., H_l |- Always P

and it is for this reason that it is recommended (see Section 2.7) that all state machine requirements be formulated in this way.

In order to apply this tactic, click on the ForwardsInduct item in the DOVE Tactics list and click on the Apply button. The effect in this case is to replace the subgoal

```
|- Always Elight = Wlight
```

with the inductive subgoal

```
Previously Always Elight = Wlight
|- Elight = Wlight
```

7.4.2 Topology

The next step is to introduce information from the state machine graph so as to allow the application of back-substitution. Click on the Topology item in the DOVE Tactics list and then on the Apply button.

The decomposition under First gives the initial case,

```
First,
At AllRed,
Previously Always Elight = Wlight
|- Elight = Wlight
```

while the projection onto Not First gives the inductive-case. The Topology tactic then determines all those begin-state, transition, final-state triples which appear in the state machine graph and are consistent with the subgoal's hypothesis list. In this case there is nothing in the hypothesis list to restrict the allowed transitions, so the tactic generates ten more subgoals, each of the same form; for example,

```
Not First,
Previously At AllRed,
At NSGreen,
By NSChangeGreen,
Previously Always Elight = Wlight
|- Elight = Wlight
```

7.4.3 Back-substitution

The final step in the proof is to introduce information about the actions performed by the various transitions (including initialisation). To do this, click on the BackSubstitute item from the DOVE Tactics list and then on the Apply to All button. The Apply to All button applies a tactic to every remaining subgoal. When applied to the initial-case subgoal, the BackSubstitute tactic introduces the initialisation condition as a hypothesis. When applied to a transition subgoal, it replaces each variable appearing in the subgoal with the value calculated by the transition's action and introduces the transition's guard as an assumption. In both cases it then attempts to apply a few simple logical rules to eliminate trivial subgoals. In this case *all* of the subgoals are trivial and the tactic is able to discharge them and the property is proved.

To see why this is so, consider the effect of back-substitution on a couple of the subgoals. The reader will find the discussion in Section 2.8 very useful here. In fact, the example given there encompasses all the technical details of the examples in this section.

In the initial-case subgoal, back-substitution simply results in the initialisation conditions being added to the hypothesis list, yielding[12] the subgoal:

```
First,
NLight = Red,
SLight = Red,
ELight = Red,
WLight = Red,
At AllRed
|- Elight = Wlight
```

This subgoal is proved by observing that both ELight and WLight have the value Red and are therefore equal. The BackSubstitute is able to recognise and discharge such a simple subgoal automatically.

In the case of the By NSChangeGreen subgoal presented in the previous subsection, the effect of applying back-substitution is:

```
At AllRed,
Elight = Wlight,
Previously Always Elight = Wlight
|- Elight = Wlight
```

In this case, the required target property appears as a hypothesis and the BackSubstitute tactic is able to discharge the subgoal automatically.

A more interesting subgoal centers on the EWChangeRed transition.

```
Not First,
Previously At EWAmber,
At AllRed,
By EWChangeRed,
```

[12] Recall that a "Previously" hypothesis is trivial in the initial configuration, so we have dropped it here.

```
Previously Always Elight = Wlight
|- Elight = Wlight
```

In this case, the simple back-substitution process yields the subgoal

```
Elight = Wlight
At EWAmber,
AmberTmOut < time, Previously Always Elight = Wlight
|- Red = Red
```

The BackSubstitute tactic is able to recognise that Red = Red and discharges the subgoal automatically.

The fact that all the subgoals are proved by the BackSubstitute tactic shows that the MasterBlast tactic will prove the property immediately. The user can verify this, by applying it directly after the ForwardsInduct step.

7.5 Advanced Tutorial: proof management in practice

This tutorial session presents a more advanced look at the support for the DOVE proof strategy via the use of Basic Tactics alongside specific DOVE tactics. The tactics which were dealt with in the previous tutorial will not be elaborated. Rather, this tutorial will concentrate on the new ideas. It is quite easy to read after completing the previous tutorial, since the more technical aspects appear in Section 7.6. The first time reader could simply follow the proof steps as a way of seeing what is available with some more effort. If there is confusion with the instructions at any point in the tutorial it may be useful to compare the steps with those given in the proof scripts in Subsection 7.6.2 below.

The fundamental properties of the TrafficLights state machine can be summarized as the colours of the lights in each state. Two such properties are proved in this tutorial. The first is the property EWAmber_NRed that specifies the colour of NLight in the state EWAmber,

```
|- Always (At EWAmber Implies NLight = Red)
```

The user should now open a property set called Tutorial in the Properties Manager, and insert the property EWAmber_NRed (as explained in Section 6.3). The proof of this property is presented as an example of introducing a “lemma” during the proof process.

That a further lemma is required can be understood from the state machine diagram in Figure 2.1, as now explained. Back-substitution drags the proof obligation back to the state EWGreen, where it decomposes under the Topology tactic to two cases.

- By EWChangeGreen, which is dragged back under the BackSubstitute tactic to the state AllRed. That the North light is Red in AllRed is manifest when LastGreen = NS, as is implied by the precondition of the transition EWChangeGreen. For, the back-substitution along NSChangeRed, and the initial predicate applicable for the First case in AllRed, obviously impose NLight = Red, while back-substitution along EWChangeRed requires LastGreen = EW and produces contradictory hypotheses.

- By WaitNS, for which the corresponding subgoal has the form

  ```
  Not First,
  Previously At EWGreen,
  At EWGreen,
  By WaitNS,
  Previously Always (At EWAmber Implies NLight = Red),
  At EWAmber Implies NLight = Red
  |- NLight = Red
  ```

  Under the BackSubstitute tactic the state returns to EWGreen. However, the hypotheses about the state EWAmber are clearly of no use in proving something about EWGreen. Hence, this subgoal cannot be directly proved by back-substitution. However, the fact that NLight is Red in the state EWGreen is manifestly correct from the definition of the TrafficLights state machine!

Thus this second item brings up the required lemma, the property EWGreen_NRed which specifies that NLight is also Red in the state EWGreen,

```
|- Always (At EWGreen Implies NLight = Red)
```

The user should also enter the property EWGreen_NRed into the Tutorial property set in the Properties Manager, and then save the state machine to permanently save the properties for later reference.

There are two ways to introduce the new property as a lemma for proof during a current proof session:

1. by entering the property into the prover via the Properties Manager (as discussed in Section 6.5) when its need arises, as an intermediate lemma requiring independent proof, and then swapping back to the original proof session and cutting in the newly proved fact using the Add Temporal Fact option under the DOVE menu;
2. by cutting in the desired fact using Add Invariant and proving it as part of the original proof session.

Both of these approaches are now outlined.

7.5.1 Intermediate lemma method

Enter the property EWAmber_NRed (defined above) into the Prover via the Properties Manager as discussed earlier (user confirmation is required), and open the current proof history via the Proof History (Current) option of the View menu.

As the property is of the "Always" form, the first step is to apply ForwardsInduct. Do this. Now bring the state information over to the hypotheses of the sequent by selecting Decompose-Sequent from the list of DOVE Tactics and then pressing Apply. Next, select Topology and press Apply. The reason for applying DecomposeSequent first, rather than immediately applying

Topology as in the previous tutorial, is that then the state information in the hypotheses limits the number of cases in the toplogy – only two transitions enter the state EWAmber– and thus greatly simplifies the proof.

The First subgoal (subgoal 3) is rather easy, since its hypotheses are clearly in contradiction[13] – if First then the state must be AllRed, but by hypothesis the state is EWAmber!

```
First,
At AllRed,
Previously Always (At EWAmber Implies NLight = Red),
At EWAmber
|- NLight = Red
```

Precisely this situation has been dealt with in SubSection 7.3.4. Make subgoal 3 the active subgoal by clicking on it with mouse button 1. From the Configuration submenu the user should apply GetConfig, followed by SolveConfig. The subgoal is then discharged.

Note that such a situation could occur quite often in such proofs, so it may be nice to make a "master tactic" which combines these two into one step. This is also a good illustration of installing User Tactics and using the tactic tree representation of the proof. Thereto, bring up the tactic tree representation at this stage by selecting the Tree Display option of the View menu. It will appear as shown in Figure 7.4.

In particular note that the third branch (from the left) ends in a black dot, indicating that this third subgoal has been discharged (in two steps). The second branch is "waiting" with a red dot, indicating that it is the active subgoal. "Cut out" the last two steps of the proof (the configuration tactics) by clicking with mouse button 2 on the node where the cut should be made – i.e., on the top (second to last) node of the third branch. The two nodes of the third branch will then turn green, and the combined tactic will be displayed in the Current Tactic frame. Now select the Install Tactic option of the Tactic menu, and write, say, the name MasterConfig in the dialog box which appears, and then press OK. The tactic is then automatically installed in the User Tactics frame.

As an example of its use, go back two steps in the proof – by selecting the Chop option of the Proof menu (twice) – to the situation after the application of the Topology tactic. Observe the behaviour of the Tactic Tree and Proof History windows. Now apply MasterConfig, noting that the third subgoal is discharged in one step.

The subgoals are now ready for back-substitution, as will be familiar from the proof of EWLightsEq in the previous tutorial. Select BackSubstitute and press Apply to All. The subgoal corresponding to the WaitTO loop is solved, while the other subgoal is dragged back to the state EWGreen.

At this point, subgoal 1 reads

```
1. MaxCars < NCars + SCars,
At EWGreen,
At EWAmber Implies NLight = Red,
```

[13] Although it will be labouring the point for readers with a mathematical background, it is otherwise worth recalling that the statement A Implies B is *true* if A is false.

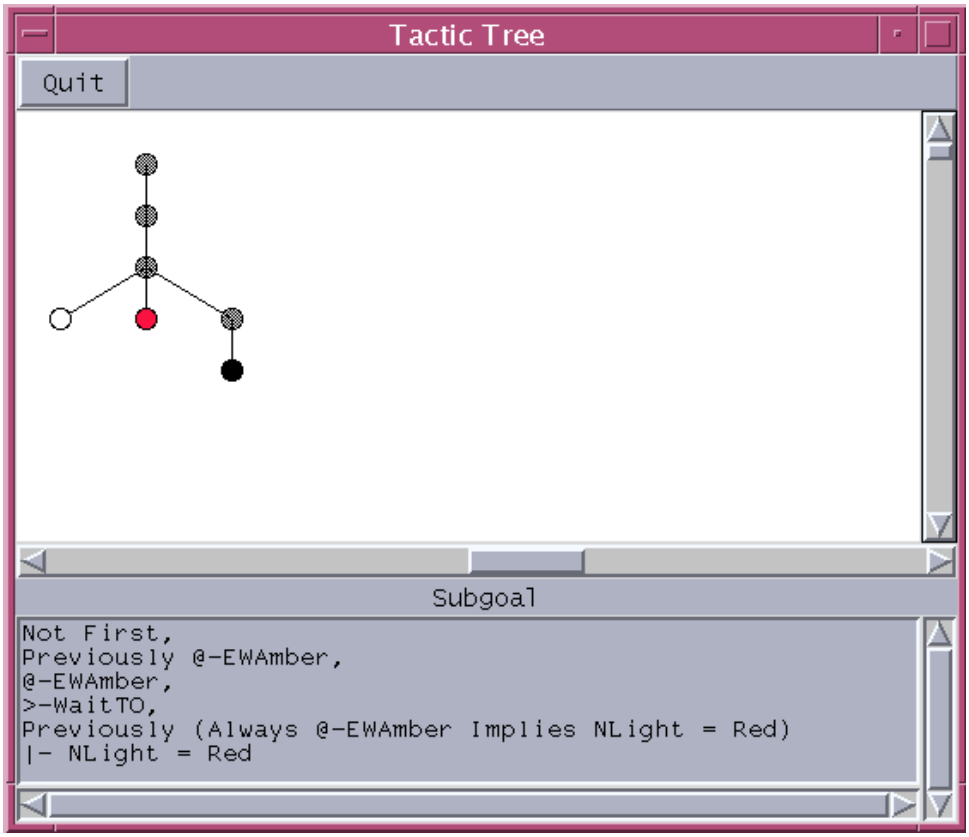


Figure 7.4: Tree Display after configuration tactics applied

```
Previously Always (At EWAmber Implies NLight = Red)
|- NLight = Red
```

The hypotheses about what `EWAmber` implies are clearly useless for proving the statement about the state `EWGreen`. In fact, they can simply be removed [14] by applying the tactic `Mach_thinL` – and rotating the hypotheses since `Mach_thinL` removes the left-most hypothesis.

To carry out the hypotheses pruning, select `ApplyTemporalRule` in the `Interactive` submenu of the `DOVE Tactics` list (by *double-clicking* with mouse button 1). The `Prover` inserts the tactic `ApplyTemporalRule_tac []` into the `Current Tactic` window and waits for the appropriate temporal rule to be inserted into the square brackets. Now put the mouse cursor into the square brackets and type in `Mach_thinL`, and then press `Apply`. The left-most hypothesis will be removed.

For an alternative means of application, as directed at the bottom of the `Prover` window, bring up the `Theorem Browser` by pressing the `Match Theorems` button (after applying `Chop` from the `Proof` menu to return to undo the last proof step). Select the `All Theorems` option of the `View`

[14] This is an *optional* step, which is applied here for two reasons: it can be good practice in long proofs to keep the hypotheses a manageable size; and the application demonstrates one way to apply `Basic Tactics`! However, as discussed below in Subsection 7.6.1, deleting inductive hypotheses can in general be very dangerous when using the DOVE back-substitution strategy. For, in general by back-substitution the proof burden will be dragged all the way back to the state where the inductive hypotheses *do* matter.

menu on the Theorem Browser, select MachCor in the Theory Names list using mouse button 1, and then select the rule to be applied (MachCor.Mach_thinL) from the Theorem Names list using mouse button 2. This will insert the theorem name into the Current Tactic window, and the user must then simply press Apply. This method can be useful in that it can allow the advanced user to peruse the matching theorems and find a useful one.

The next hypothesis is the state information we want to keep, so we first rotate it to the end by applying (in the same way) the temporal rule Mach_exchange. (Actually, the user can simply delete the content of the square brackets in the Current Tactic window and replace it with Mach_exchange, and then press Apply.) Now apply Mach_thinL twice more, at which point the subgoal is simply

```
1. At EWGreen |- NLight = Red
```

Clearly this is a new result which must be proved. In fact, it is a consequence of the property EWGreen_NRed (defined above). The strategy now is to first prove the property EWGreen_NRed, and then to use this fact to prove the remaining subgoal. So, enter the property EWGreen_NRed into the Prover via the Properties Manager as usual. DOVE asks the user to confirm that a new proof is required, and then loads in the property to be proved. Note that the Proof History window containing the current proof history changes to be that of EWGreen_NRed.

The proof of this property goes along the same lines as the above. In fact, the user should follow essentially the same steps without bothering to thin the hypotheses, until the remaining subgoal reads

```
1. LastGreen = NS,
At AllRed,
At EWGreen Implies NLight = Red,
Previously Always (At EWGreen Implies NLight = Red)
|- NLight = Red
```

Now that the proof burden has been dragged back to the AllRed state, the now-familiar method solves it: apply Topology, and then select BackSubstitute and press Apply to All. After completing this step DOVE greets the user with the message that EWGreen_NRed is proved. The user should press OK, at which point the ML image is updated to contain the new theorem. The user should save the proof of EWGreen_NRed into a proof script file using the Save option of the File menu.

To now return to the proof of EWAmber_NRed, select it under the Swap menu. The fact which has just been proved can be "cut in" to the hypotheses of subgoal 1 by using the tactic AddFact_tac. To do this, select the Add Temporal Fact option[15] in the Interactive submenu of the DOVE Tactics list (by *double-clicking* with mouse button 1). The Prover inserts the tactic AddFact_tac [] into the Current Tactic window and waits for the appropriate temporal fact to be inserted into the square brackets. Now put the mouse cursor into the square brackets

[15]It may strike the user as strange that the theorem EWGreen_NRed is not applied to prove the subgoal in the same way as, say, thinL was used above. The point is that – while it certainly contains the information required to prove the subgoal – it is not of the precise form to pattern match on the form of the subgoal. The tactic Add Temporal Fact takes care of this simple problem.

and type in Tutorial_EWGreen_NRed, and then press Apply. The alternative application through the Theorem Browser, as discussed above, can also be used.

From the form of the subgoal

```
1. At EWGreen,
Always (At EWGreen Implies NLight = Red)
|- NLight = Red
```

it is now clear that the proof is essentially finished. However, it is necessary to first extract the current state information from the "Always" hypothesis. This is the result of another rule, Mach_HasAlwaysL, in the MachCor theory. Apply it[16] via the tactic ApplyTemporalRule as described above. The result is

```
1. At EWGreen,
At EWGreen Implies NLight = Red
Previously Always (At EWGreen Implies NLight = Red)
|- NLight = Red
```

This final subgoal can be discharged by applying SolveTemporal from the DOVE Tactic list.

Again the user is told that the proof has been completed, and must press OK. At this point the user should save the proof into a proof script file using the Save option of the File menu on the Prover. It is also good management to save the state machine via the Save option of the File menu of the Edit window. This records the status of the properties as being proved.

7.5.2 Add Invariant method

Now re-enter the property EWAmber_NRed into the Prover via the Properties Manager as usual. The beginning of this proof follows precisely the steps given above, so the user should repeat them and stop just before loading the property EWGreen_NRed into the Prover, at which point the subgoal to be proved is

```
1. At EWGreen |- NLight = Red
```

Instead of introducing the desired proof as a new property, it may be introduced as a fact to be proved in the current proof. To do this, select AddInvariant from the DOVE menu[17] (by *double-clicking* with mouse button 1). It will save the user time if the text of subgoal 1 is first put into the clipboard buffer. Then it may be inserted into the window brought up by Add Invariant, and edited[18] to the form

[16] To understand more about when the pattern-matching to the subgoal will work, the user should examine Subsection 7.6.3.

[17] For a more general discussion of the function of the Add Invariant tactic, see Subsection 7.6.1.

[18] The need for this editing will not be immediately obvious to the user, since again it is simply to ensure the correct syntax given the temporal sequent structure of the DOVE deductive system. The invariant must be stated as a temporal formula, not a sequent. Although the translation is a "triviality" for the user, given that pattern-matching is at the basis of proof tools it must be done explicitly in DOVE.

```
At EWGreen Implies NLight = Red.
```

On pressing OK, this statement is inserted into the hypotheses of the subgoal, and a further subgoal is created which enforces that the statement be proved to be true. At this point then, the proof state is

```
1. At EWGreen,
At EWGreen Implies NLight = Red,
Previously Always (At EWGreen Implies NLight = Red)
|- NLight = Red
2. At EWGreen |-
Always (At EWGreen Implies NLight = Red)
```

Apply SolveTemporal to solve subgoal 1.

The proof should now be completed by following the steps used to prove EWGreen_NRed in the previous subsection. A convenient way to achieve this is to bring up the proof history for that proof, via the option Proof History (Other) of the View menu. Then desired proof steps can be applied simply by double-clicking on the proof history (taking account of the advice in E.2.3).

Again, on completion the user is told that the proof has been completed and must press OK. At this point the user should save the proof into a proof script file – as the previous proof of EWAmber_NRed has been saved into the file Tutorial_EWAmber_NRed.prf, it is best to use the Save As option of the File menu to save under a different name, such as Tutorial_EWAmber_NRed_new.prf.

7.5.3 Using MasterBlast

The above proofs give quite a good overview of the DOVE proof strategy. Recall that (an extension of) part of this strategy has been automatically encoded in the MasterBlast tactic: namely, the Topology tactic followed by repeated applications of BackSubstitute and various automatic solution tacticals. Not surprisingly, using this tactic gives a much shorter proof. The user may wish to redo the proofs of these properties while incorporating the MasterBlast tactic. If desired the proof script can be saved. It is best to use the Save As option of the File menu to save under a new name, such as Tutorial_EWAmber_NRed_MB.prf.

Since the structure of the proof is somewhat similar, it is not necessary to give detailed explanations of how to do this. The user should consult the appropriate proof scripts in Subsection 7.6.2 if there is any uncertainty in how to proceed. It is interesting to see how the application of MasterBlast at the third proof step in the proof of EWAmber_NRed immediately leads to the decision point for introducing the new invariant EWGreen_NRed. Also, note that the optional steps for cleaning up the hypotheses prior to introducing the new invariant have not been implemented in the proof script Tutorial_EWAmber_NRed_MB.prf.

7.6 Methods used in the Advanced Tutorial

In this section the methods used in the advanced tutorial are elaborated. This should help in understanding both the context in which they are used, as well as how they are used.

7.6.1 Keeping invariants using `Add Invariant`

In the above Advanced Tutorial, `Add Invariant` simply provided another means of introducing a required fact and establishing its proof. More generally, the benefit of `Add Invariant` is that it gives a straightforward way to add new facts while at the same time keeping the known facts which may be used later along the back-substitution path. In keeping with the overall DOVE strategy, its use can be explained in the following way. The proof of a given property is establishing an invariant of the system. As the proof proceeds it may be found that other invariants are required, and must also be established as part of the overall proof. `Add Invariant` keeps all the invariants manifest in the hypotheses while constructing further subgoals corresponding to their proof requirement.

A scenario where this construction is essential would arise if the `LastGreen` variable was not used in the TrafficLights model. The value of `LastGreen`, as inserted in back-substitution through a transition precondition, was crucial in allowing the proof strategy to terminate in the `AllRed` state. If `LastGreen` was not used then the proof would not terminate at all!

However, in this case consider the property `AllRed_NRed` that specifies the colour of the North light in the state `AllRed`,

```
|- Always (At AllRed Implies NLight = Red)
```

Since the `LastGreen` variable is not included, back-substitution goes through both of the `EWChangeGreen` and `NSChangeRed` transitions. The latter is clearly trivial. But, as in the Advanced Tutorial, in the `EWChangeGreen` path new invariants corresponding to the colour of the North light in `EWAmber` and `EWGreen` must be included. Using `Add Invariant` to include them keeps the inductive hypotheses

```
At AllRed Implies NLight = Red,
Previously Always (At AllRed Implies NLight = Red)
```

in the hypotheses list of the corresponding subgoals. This is crucial since by back-substitution along this path the proof obligation returns to the `AllRed` state.

The user will note that this example also ties in nicely with the previous discussion warning against careless pruning of inductive assumptions. For here it is essential to keep the hypotheses if this proof procedure is to work.

7.6.2 Proof scripts

The proof scripts of the above proofs were deposited in files: into `EWAmber_NRed.prf` and `EWGreen_NRed.prf` for the concurrent proof; and into `EWAmber_NRed_new.prf` for the proof using `Add Invariant`. The user may like to look at these files. The proof scripts for the concurrent proof are

```
by (ForwardsInduct 1);
by (DecomposeSequent 1);
by (Topology 1);
```

```
by (GetConfig 3);
by (SolveConfig 3);
by (BackSubstitute 1);
by (BackSubstitute 2);
by (ApplyTemporalRule_tac [Mach_thinL] 1);
by (ApplyTemporalRule_tac [Mach_exchange] 1);
by (ApplyTemporalRule_tac [Mach_thinL] 1);
by (ApplyTemporalRule_tac [Mach_thinL] 1);
by (Mach_AddFact_tac [Tutorial_EWGreen_NRed] 1);
by (ApplyTemporalRule_tac [Mach_HasAlwaysL] 1);
by (SolveTemporal 1);
qed Tutorial_EWAmber_NRed;
```

and

```
by (ForwardsInduct 1);
by (DecomposeSequent 1);
by (Topology 1);
by (MasterConfig 3);
by (BackSubstitute 1);
by (BackSubstitute 2);
by (Topology 1);
by (BackSubstitute 1);
by (BackSubstitute 1);
by (BackSubstitute 1);
qed Tutorial_EWGreen_NRed;
```

The "1" in, for example, "`by (ForwardsInduct 1)`" denotes that this step is applied to subgoal 1. It is automatically inserted into the proof script as determined by which subgoal is being worked on (i.e., by which subgoal has been highlighted red by clicking with mouse button 1). Also, the lower subgoals are automatically renumbered when a given subgoal is discharged. That explains why, for example, `by (BackSubstitute 1)` appears three times in a row at the end of the proof of `EWGreen_NRed`.

The proof history for the proof using `Add Invariant` is

```
by (ForwardsInduct 1);
by (DecomposeSequent 1);
by (Topology 1);
by (MasterConfig 3);
by (BackSubstitute 1);
by (BackSubstitute 2);
by (ApplyTemporalRule_tac [Mach_thinL] 1);
by (ApplyTemporalRule_tac [Mach_exchange] 1);
by (ApplyTemporalRule_tac [Mach_thinL] 1);
by (ApplyTemporalRule_tac [Mach_thinL] 1);
by ((add_invariant_tac "@-EWGreen Implies (NLight = Red)") 1);
by (SolveTemporal 1);
```

```
by (ForwardsInduct 1);
by (DecomposeSequent 1);
by (Topology 1);
by (MasterConfig 3);
by (BackSubstitute 1);
by (BackSubstitute 2);
by (Topology 1);
by (BackSubstitute 1);
by (BackSubstitute 1);
by (BackSubstitute 1);
qed Tutorial_EWAmber_NRed_new;
```

The proof history for the `Add Invariant` proof using `MasterBlast` is

```
by (ForwardsInduct 1);
by (DecomposeSequent 1);
by (MasterBlast 1);
by ((add_invariant_tac "@-EWGreen Implies (NLight = Red)") 1);
by (SolveTemporal 1);
by (ForwardsInduct 1);
by (DecomposeSequent 1);
by (MasterBlast 1);
by (MasterBlast 1);
qed Tutorial_EWAmber_NRed_MB;
```

7.6.3 A brief look at the `Basic Tactics` used

In this last subsection the various `Basic Tactics` which have (or could have) been used in the Advanced Tutorial are briefly discussed. The informal explanations here should be considered in conjunction with the results obtained when using these rules at intermediate stages of the proofs. In these explanations we use the "natural deduction" syntax, as introduced in Section 7.1.2, whereas the rules are listed in Appendix C in standard Isabelle syntax. Thus, the discussion here should also help in understanding that Appendix.

Consider the passage of steps which pruned useless hypotheses from the hypotheses list, using the theorems `Mach_exchange` and `Mach_thinL`. As given in Appendix C, these rules[19] are

$$\texttt{Mach\_exchange}: \frac{\texttt{LeftExtending G} \quad \texttt{LeftExtending H} \quad \texttt{\$G, \$H, Q, P |- R}}{\texttt{\$G, P, \$H, Q |- R}}$$

and

$$\texttt{Mach\_thinL}: \frac{\texttt{LeftExtending } H \quad \texttt{\$}H, \texttt{\$}G \texttt{ |- P}}{\texttt{\$}H, R, \texttt{\$}G \texttt{ |- } P}$$

[19] Here `$G`, e.g., stands for an arbitrary hypothesis list of properties, while `P` (without the $) is a single property.

Recall that they are applied in a goal-directed (“backwards”) proof system. Thus, when matching to a given subgoal the user should match the RHS of the rule (i.e., the RHS of the meta-implication ==>) to the subgoal. Correspondingly, in the natural deduction syntax the user should match to “below the line”. The result of applying the rule will then be given by its LHS (one subgoal for each entry separated by semicolon) – “above the line” for the natural deduction syntax – with the substitutions which allowed the matching. Thus, up to the LeftExtending part, Mach_exchange simply rotates the first hypothesis to be last, and Mach_thinL deletes the first hypothesis from the hypotheses list.

The LeftExtending subgoals never appear in the subgoals listed in the Prover window, as the user will have discovered earlier in the tutorial. This is because the tacticals which apply the rules, such as ApplyTemporalRule, have automatically solved them before returning the result. Indeed, they are simply technical baggage needed in the modelling of sequents by hypotheses lists in a way which is convenient for unifying with general subgoals. The advanced user will realise that discharging these LeftExtending subgoals is what distinguishes, say, ApplyTemporalRule from Resolve (Intro) of the Basic Tactics list. From now on we will ignore LeftExtending subgoals in discussions of the rules.

Now consider the rule Mach_HasAlwaysL, which was used in the concurrent proof,

$$\texttt{Mach\_HasAlwaysL}\colon \frac{\texttt{LeftExtending}\ E \qquad \$E,\ P,\ \texttt{Previously}(\texttt{Always}\ P),\ \$F\ \texttt{|-}\ R}{\$E,\ \texttt{Always}\ P,\ \$F\ \texttt{|-}\ R}$$

Again reading right to left, this simply breaks up a fact which is always true into the corresponding fact for the current state, and the statement that the fact is true at all previous times.

As a more advanced observation, note that the tactic DecomposeSequent has been used to take, for example,

1. First |- At EWAmber Implies NLight = Red

to

1. First, At EWAmber |- NLight = Red

This is a consequence of the DOVE rule Mach_impR which is used in the DecomposeSequent tactical,

$$\texttt{Mach\_impR}\colon \frac{\texttt{LeftExtending}H \qquad \$H,\ P\ \texttt{|-}\ Q}{\$H\ \texttt{|-}\ P\ \texttt{Implies}\ Q}$$

It may sometimes be desirable to use this rule, particularly if the user wants to concentrate on one subgoal where the target goal has several conjunctions in the conclusion of an implication. For DecomposeSequent also contains the rule

$$\texttt{Mach\_conjR}\colon \frac{\$H\ \texttt{|-}\ P \qquad \$H\ \texttt{|-}\ Q}{\$H\ \texttt{|-}\ P\ \texttt{And}\ Q}$$

which breaks up a set of target conjunctions into separate subgoals. However, given the facility `Apply to All`, this may simply be a matter of taste.

Also, the tactic `SolveTemporal` has been used to solve subgoals of the form

1. `At EWGreen, At EWGreen Implies NLight = Red |- NLight = Red`

The user will now observe that this is a consequence of the rule

$$\texttt{Mach\_impL}: \frac{\texttt{LeftExtending}\,H \qquad \$H, \$G \mid\!- P \qquad \$H, Q, \$G \mid\!- R}{\$H, P\ \texttt{Implies}\ Q, \$G \mid\!- R}$$

followed by two applications of the rule

$$\texttt{Mach\_basic}: \frac{\texttt{LeftExtending}\,H}{\$H, P, \$G \mid\!- P}$$

These rules are included in the `SolveTemporal` tactical. Applying `Mach_impL` produces two subgoals which say that it is enough to prove the antecedent of the implication, and then its conclusion can be used as a hypothesis,

1. `At EWGreen |- At EWGreen`

2. `At EWGreen, NLight = Red |- NLight = Red`

The manifestly obvious rule, `Mach_basic`, proves both of these subgoals.

It may be that these examples will help the user to explore the use of other rules in AppendixC.

7.6.4 Applying constant definitions via `Basic Tactics`

Via the `Rule Definition` option of the `Definitions` menu, the user is able to insert definitions of the constants which are being declared in the design of a given state machine. It is generally expected that the rules defined really are *definitions*, in the sense of equalities which should be applied by rewriting of the constants. Examples of such have been discussed in Section 4.5.4.2. In proving properties in the state machine theory, the user may then need to apply these rewriting rules to unwind the definitions. In this section we explain the mechanism for applying the rewriting rules in XIsabelle, using an example from the TrafficLights theory.

Assume that we have defined `MaxCars` by the rule `MC_three` (see Section 4.5.4.2), and consider the property

```
|- Always (((Previously At EWGreen) And (Previously #3 < NCars)) Implies
Not ELight = Green) .
```

This is a very trivial example of the use of rewriting, but serves to illustrate the point. After applying `ForwardsInduct`, `DecomposeSequent`, and then `MasterBlast`, the only remaining subgoal is (having pruned away the irrelevant hypotheses)

```
1. Not (MaxCars < NCars + SCars),
#3 < NCars
|- False
```

Here we would like to apply the rule MC_three to obtain a contradiction of hypotheses.

To do this, select the Simplify... option in the Basic Tactics frame, and click once on Simplify Subgoal with mouse button 1. The Prover inserts the tactic simp_tac (simpset() addsimps []) into the Current Tactic window and waits for the appropriate rewrite rule to be inserted into the square brackets. Now, as directed at the bottom of the Prover window, bring up the Theorem Browser by pressing the Match Theorems button. Select the Axioms option of the View menu on the Theorem Browser, select TrafficLights in the Theory Names list using mouse button 1, and then select the rule to be applied (TrafficLights.MC_three) from the Theorem Names list using mouse button 2. This will insert the theorem name into the Current Tactic window, and the user must then simply press Apply. The results is to unwind the definition of MaxCars, leaving

```
1. Not(#3 < NCars + SCars),
#3 < NCars
|- False
```

The user then needs simply apply SolveTemporal to discharge the subgoal. [Note that this is a configuration subgoal, so the usual GetConfig then SolveConfig tactic is what is actually discharging it.]

References

1. Australian Department of Defence. *Def(Aust) Standard 5679: The procurement of Computer Based Safety Critical Systems*, 1998. iii

2. Cambridge University, `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/docs.html`. *The XIsabelle User Manual*, 1998. 4, 61, 66, 68, 76, 80, 81, 110

3. European Communities – Commission. *ITSEC: Information Technology Security Evaluation Criteria*, 1991. iii, 77

4. E. Kindler. Safety and liveness properties: A survey. *EATCS Bulletin*, 53, June 1994. 19

5. D. Libes. *Exploring Expect*. O'Rielly and Associates, 1995. 4

6. J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994. 4

7. L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1996. 4

8. L. C. Paulson. *The Isabelle Reference Manual*. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/docs.html`, 2000. 80

9. L. C. Paulson and T. Nipkow. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994. 4, 9, 68, 76

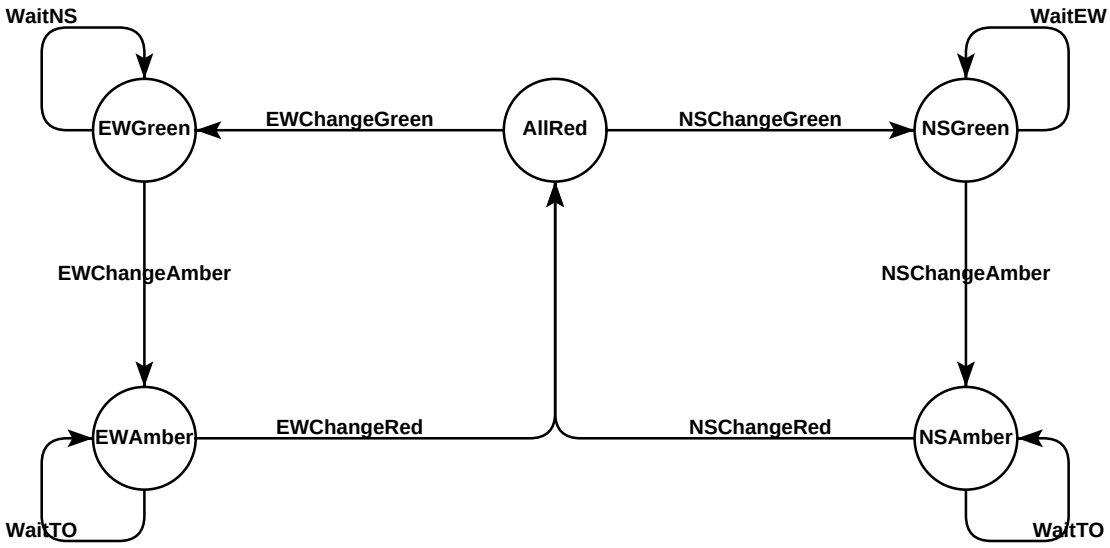
10. S. M. Rubin. Computer aids for VLSI design. `http://www.staticfreesoft.com/documentsTextbook.html`. 4

11. L. C. Paulson T. Nipkow and M. Wenzel. *Isabelle's Logics: HOL*. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/docs.html`, 2000. 35, 110

12. S. Berghofer T. Nipkow. *The Isabelle System Manual*. Technische Universität München, `http://www.cl.cam.ac.uk/Research/HVG/Isabelle/docs.html`, 2000. 133

13. UK Ministry of Defence. *Defence Standard 00-55: The procurement of Safety Critical Software in Defence Equipment*, 1995. iii

14. UK Ministry of Defence. *Defence Standard 00-56: The procurement of Safety Critical Software in Defence Equipment*, 1995. iii



Appendix A The TrafficLights state machine

theory *TrafficLights = DOVE:*

startmachine

states *AllRed | EWAmber | EWGreen | NSAmber | NSGreen*

datatype *Colour = "Red" | "Amber" | "Green"*

datatype *Direction = "EW" | "NS"*

consts
"MaxCars" :: "nat"
"MaxTime" :: "nat"

inputs
"ECars" :: "nat"
"NCars" :: "nat"
"SCars" :: "nat"
"WCars" :: "nat"
"time" :: "nat"

heaps
"AmberTimeOut" :: "nat"
"ELight" :: "Colour"
"LastGreen" :: "Direction"

```
  "NLight" :: "Colour"
  "SLight" :: "Colour"
  "WLight" :: "Colour"

initpred
  "(NLight = Red) & (SLight = Red) & (ELight = Red) & (WLight = Red)"

transdef "NSChangeGreen" "
  Guard: LastGreen = EW
  Act: NLight <-- Green;
       SLight <-- Green;"

transdef "EWChangeRed" "
  Guard: AmberTimeOut < time
  Act: ELight <-- Red;
       WLight <-- Red;
       LastGreen <-- EW;"

transdef "EWChangeAmber" "
  Let: NSnum <-- (NCars + SCars);
  Guard: MaxCars < NSnum
  Act: ELight <-- Amber;
       WLight <-- Amber;
       AmberTimeOut <-- (time + MaxTime);"

transdef "NSChangeAmber" "
  Let: EWnum <-- (ECars + WCars);
  Guard: MaxCars < EWnum
  Act: NLight <-- Amber;
       SLight <-- Amber;
       AmberTimeOut <-- (time + MaxTime);"

transdef "NSChangeRed" "
  Guard: AmberTimeOut < time
  Act: NLight <-- Red;
       SLight <-- Red;
       LastGreen <-- NS;"

transdef "EWChangeGreen" "
  Guard: LastGreen = NS
  Act: ELight <-- Green;
       WLight <-- Green;"

transdef "WaitEW" "
  Let: EWnum <-- (ECars + WCars) ;
  Guard: Not (MaxCars < EWnum)
  Act: Skip;"
```

```
transdef "WaitNS" "
  Let: NSnum <-- (NCars + SCars) ;
  Guard: Not (MaxCars < NSnum)
  Act: Skip;"

transdef "WaitTO" "
  Guard: Not (AmberTimeOut < time)
  Act: Skip;"


graph "AllRed" "
    EWGreen --EWChangeAmber--> EWAmber
  | AllRed --EWChangeGreen--> EWGreen
  | EWAmber --EWChangeRed--> AllRed
  | NSGreen --NSChangeAmber--> NSAmber
  | AllRed --NSChangeGreen--> NSGreen
  | NSAmber --NSChangeRed--> AllRed
  | NSGreen --WaitEW--> NSGreen
  | EWGreen --WaitNS--> EWGreen
  | EWAmber --WaitTO--> EWAmber
  | NSAmber --WaitTO--> NSAmber"

endmachine

end
```

Appendix B Syntax of DOVE

In using the DOVE tool, the user must be able to write a transition definition in designing the state machine, and to write a sequent in specifying temporal properties. The following grammar trees summarizes the concrete syntax which the user must employ for these purposes. In these grammars, we have used a number of notational devices:

- *id* is an identifier; i.e., a string of alphanumeric characters and underscore which must begin with a letter;
- *idts* stands for a list of identifiers with optional specification of type;
- *type* stands for any Isabelle type available in the state machine theory;
- *expr* stands for any mathematical expression (type-correct Isabelle term) in the variables and constants available in the state machine theory, and from any included theories; and
- where needed to disambiguate, the operator arguments are specified.

B.1 Transition definition

transition definition	⟶	let_expression
		guard_part
		action_part
let_expression	⟶	‘’
	|	LET assignmentlist
guard_part	⟶	‘’
	|	GUARD *expr*
action_part	⟶	‘’
	|	ACTION assignmentlist
assignmentlist	⟶	*id* GETS *expr*‘;’
	|	assignmentlist *id* GETS *expr*‘;’
ACTION	⟶	‘Act:’
GETS	⟶	‘<--’
GUARD	⟶	‘Guard:’
LET	⟶	‘Let:’

B.2 Sequent

sequent	⟶	formulalist TURNSTILE formula
	|	TURNSTILE formula
formulalist	⟶	formula

	|	formulalist ‘,’ formula
formula	⟶	formula AND formula
	|	formula OR formula
	|	formula IMPLIES formula
	|	formula CONGRUENCE formula
	|	formula FROMTHENON formula
	|	formula FROMTHENONS formula
	|	MOSTRECENTLY formula formula
	|	MOSTRECENTLYS formula formula
	|	FORALL *idts* . formula
	|	EXISTS *idts* . formula
	|	ALWAYS formula
	|	SOMETIME formula
	|	formula
	|	NOT formula
	|	PREVIOUSLY formula
	|	PREVIOUSLYS formula
	|	INITIALLY formula
	|	AT *id*
	|	BY *id*
	|	FIRST
	|	TRUE
	|	FALSE
	|	*expr*

ALWAYS	⟶	‘`[-]`’
	|	‘Always’
AND	⟶	‘&’
	|	‘And’
AT	⟶	‘`@-`’
	|	‘At’
BY	⟶	‘`>-`’
	|	‘By’
CONGRUENCE	⟶	‘`<-->`’
	|	‘EquivalentTo’
EXISTS 1_ . 2_	⟶	‘`? 1_ . 2_`’
	|	‘Exists 1_ . 2_’
FALSE	⟶	‘False’
FIRST	⟶	‘first’
	|	‘First’
FORALL 1_ . 2_	⟶	‘`! 1_ . 2_`’
	|	‘ForAll 1_ . 2_’
FROMTHENON	⟶	‘`~~>`’
	|	‘FromThenOn’
FROMTHENONS	⟶	‘`~~>s`’
	|	‘FromThenOnS’
IMPLIES	⟶	‘`-->`’

	|	‘Implies’
INITIALLY	⟶	‘init’
	|	‘Initially’
MOSTRECENTLY 1_ 2_	⟶	‘<(1_) 2_’
	|	‘MostRecently 1_ 2_’
MOSTRECENTLYS 1_ 2_	⟶	‘<S(1_) 2_’
	|	‘MostRecentlyS 1_ 2_’
NOT	⟶	‘~’
	|	‘Not’
OR	⟶	‘|’
	|	‘Or’
PREVIOUSLY	⟶	‘(-)’
	|	‘Previously’
PREVIOUSLYS	⟶	‘(S)’
	|	‘PreviouslyS’
SOMETIME	⟶	‘<->’
	|	‘Sometime’
TRUE	⟶	‘True’
TURNSTILE	⟶	‘|-’

Appendix C Rules of temporal logic

The DOVE prover is a specialised version of the XIsabelle theorem prover interface. A general introduction to the use of the DOVE Prover may be found in Chapter 7 and a detailed explanation of the XIsabelle interface may be found in the XIsabelle user manual [2]. This appendix presents a list of the logical rules introduced by the DOVE tool for reasoning about temporal properties in state machines. Discussion of general logical rules specific to HOL, not DOVE, may be found in the reference manual *Isabelle's Logics: HOL* [11] in the Isabelle distribution.

The rules are divided into two categories.

- The *structural* rules are those rules which are likely to appear as matching rules in the 'Intr Rules' box. These rules are presented in Section C.1.
- The *rewriting* rules are a collection of equalities between temporal logic formulae. These rules are presented in Section C.2.

Within these categories the rules are sorted on the name of the rule according to lexicographical ASCII ordering, this means that all capitalised names come before any uncapitalised names.

In comparison to the representation in 7.6.3 the reader will notice that, as written here, the rules are somewhat less "friendly". The listing here is in the syntax used by Isabelle. The translation is straightforward: e.g., from

$$\frac{P_1 P_2}{C},$$

to

```
[|P1; P2|] ==> C
```

or vice-versa, as required. Moreover, Isabelle puts "?" in front of any free variables in the theorem – so we have included them in this appendix.

C.1 Structural rules

The structural rules are used to break up the structure of complex logical formulae. It is frequently a good idea to eliminate as many of the temporal operators as possible by application of the various structural rules. For this reason, a brief explanation is given of the proof purpose of each of the structural rules. Note that the `LeftExtending` hypotheses are technical baggage, as discussed in Section 7.6.3, and can be ignored on reading.

Some of the simpler rules are bundled into the simplification tactics in the 'DOVE Tactics' box.

Rule	Explanation

`Mach_ExistsL:`
`  [| LeftExtending ?H1;`
`  !!x. $?H1, ?P1 x, $?G1 |- ?E1 |]`
`  ==> $?H1, Exists x. ?P1 x, $?G1 |- ?E1`
	split an existentially quantified hypothesis

`Mach_ExistsR:`
`  $?H |- ?P ?x`
`  ==> $?H |- Exists x. ?P x`
	split an existentially quantified goal

`Mach_FALSEL:`
`  LeftExtending ?H`
`  ==> $?H, FALSE, $?G |- ?R`
	a false hypothesis proves anything

`Mach_ForAllL:`
`  [| LeftExtending ?H1;`
`  $?H, ?P ?x, $?G |- ?E |]`
`  ==> $?H, ForAll x. ?P x, $?G |- ?E`
	split a universally quantified hypothesis

`Mach_ForAllR:`
`  !!x. $?H1 |- ?P1 x`
`  ==> $?H1 |- ForAll x. ?P1 x`
	split a unversally quantified goal

`Mach_FTOinduct_forward:`
`  (?P,?P FromThenOn ?RR' |- ?RR) -->`
`(Not ?P, Previously ?RR, ?P FromThenOn ?RR' |- ?RR)`
`  --> (?P FromThenOn ?RR' |- ?P FromThenOn ?RR)`
	reasoning forwards to prove fromthenon

`Mach_HBInductForward_noinit:`
`  [| LeftExtending ?GA;`
`  Shbstar (?G []) (?GA []);`
`  $?GA, Previously (Always ?R) |- ?RR |]`
`  ==> $?G |- Always ?R`
	prove always goal by working forward

Rule	Explanation
Mach_HasAlwaysL: [| LeftExtending ?E; $?E, ?P, Previously Always ?P, $?F |- ?R |] ==> $?E, Always ?P, $?F |- ?R	split always hypothesis
Mach_HasAlwaysR: [| Shbstar (?E []) (?E1 []); $?E1 |- ?P |] ==> $?E |- Always ?P	use any Always hypotheses to prove an Always goal, the [-]> operator selects the Always hypotheses
Mach_Initial_removeL: [| LeftExtending ?G; LeftExtending ?H; $?G, ?q, $?H, First |- ?p |] ==> $?G, Initially ?q, $?H, First |- ?p	split an initial state hypothesis
Mach_Initial_removeR: [| LeftExtending ?G; $?G, First |- ?p |] ==> $?G, First |- Initially ?p	split an initial state goal
Mach_PredFALSE: [| Spredstar (?E []) ?Q; ! c. ?Q c = FalseV |] ==> $?E |- ?P	contradictory configuration hypotheses prove any goal
Mach_PredLR: [| Spredstar (?E []) ?Q; ! c. ?Q c --> ?P c |] ==> $?E |- ?P	prove a configuration goal by collecting configuration hypotheses
Mach_TRUER: $?H |- TrueV	anything proves a true goal

Rule	Explanation
Mach_back_fto: [| ?G |- ?RR; |- ?RR And Previously Not ?RR Implies ?P |] ==> ?G |- ?P FromThenOn ?RR	reasoning backwards to prove fromthenon
Mach_back_ftos: [| ?G |- ?RR; |- First Implies Not ?RR; |- ?RR And Previously Not ?RR Implies ?P |] ==> ?G |- ?P FromThenOnS ?RR	reasoning backwards to prove fromthenons
Mach_back_sometime: [| ?G |- ?RR; |- First Implies Not ?RR; |- ?RR And Previously Not ?RR Implies ?P |] ==> ?G |- Sometime ?P"	reasoning backwards to prove sometime
Mach_basic: LeftExtending ?H ==> $?H, ?P, $?G |- ?P	prove a goal by hypothesis
Mach_conjL: [| LeftExtending ?H; $?H, ?P, ?Q, $?G |- ?R |] ==> $?H, ?P And ?Q, $?G |- ?R	split a conjoined hypothesis
Mach_conjR: [| $?H |- ?P; $?H |- ?Q |] ==> $?H |- ?P And ?Q	split a conjoined goal

Rule	Explanation
Mach_contract: [| LeftExtending ?G1; LeftExtending ?G2; $?G1, ?P, $?G2, $?G3 |- ?R |] ==> $?G1, ?P, $?G2, ?P, $?G3 |- ?R	remove duplicate hypotheses
Mach_cut: [| LeftExtending ?G; SCovered (?G []) (?H []); $?H, ?P |- ?R; $?G |- ?P |] ==> $?H |- ?R	prove goal R via intermediate goal P.
Mach_disjL: [| LeftExtending ?H; $?H, ?P, $?G |- ?R; $?H, ?Q, $?G |- ?R |] ==> $?H, ?P Or ?Q, $?G |- ?R	split a disjoined hypothesis
Mach_disjR1: $?H |- ?P ==> $?H |- ?P Or ?Q	prove the first part of a disjunctive goal
Mach_disjR2: $?H |- ?Q ==> $?H |- ?P Or ?Q	prove the second part of a disjunctive goal
Mach_exchange: [| LeftExtending ?G; LeftExtending ?H; $?G, $?H, ?Q, ?P |- ?R|] ==> $?G, ?P, $?H, ?Q |- ?R	change the order of hypotheses

Rule	Explanation
Mach_fto_extend: [| ?G |- ?P FromThenOn ?RR; op # ?P |- ?P' FromThenOn ?RR |] ==> ?G |- ?P' FromThenOn ?RR	fromthenon extends by implication
Mach_fto_mono: [| ?G |- ?P FromThenOn ?RR; |- ?RR Implies ?RR' |] ==> ?G |- ?P FromThenOn ?RR'	fromthenon is monotonic
Mach_fto_pre_mono: [||- ?P Implies ?P'; ?G |- ?P FromThenOn ?RR |] ==> ?G |- ?P' FromThenOn ?RR	fromthenon is monotonic
Mach_ftos_extend: [| ?G |- ?P FromThenOnS ?RR; op # ?P |- ?P' FromThenOnS ?RR |] ==> ?G |- ?P' FromThenOnS ?RR	fromthenons extends by implication
Mach_ftos_mono: [| ?G |- ?P FromThenOnS ?RR; |- ?RR Implies ?RR' |] ==> ?G |- ?P FromThenOnS ?RR'	fromthenons is monotonic
Mach_ftos_pre_mono: [||- ?P Implies ?P'; ?G |- ?P FromThenOnS ?RR |] ==> ?G |- ?P' FromThenOnS ?RR	fromthenons is monotonic
Mach_identity: $?G, ?P, $?H |- ?R ==> $?G, ?P, $?H |- ?R	this may arise as a result of a cut application

Rule	Explanation
Mach_impL: [| LeftExtending ?H; $?H, $?G |- ?P; $?H, ?Q, $?G |- ?R |] ==> $?H, ?P Implies ?Q, $?G |- ?R	split a conditional hypothesis
Mach_impR: [| LeftExtending ?H; $?H, ?P |- ?Q |] ==> $?H |- ?P Implies ?Q	split a conditional goal
Mach_mp_like: [| |- ?RR Implies ?RR'; $?G |- ?RR |] ==> $?G |- ?RR'	example of "cutting"
Mach_notL: [| LeftExtending ?H; $?H, $?G |- ?P |] ==> $?H, Not ?P, $?G |- FalseV	split a negated hypothesis
Mach_notR: [| LeftExtending ?H; $?H, ?P, $?G |- FalseV |] ==> $?H, $?G |- Not ?P	split a negated goal
Mach_small_ftoR: |- ?RR ==> |- ?P FromThenOn ?RR	split a fromthenon goal

Rule	Explanation
Mach_sometime_mono: [| ?G |- ?P Sometime ?RR; |- ?RR Implies ?RR' |] ==> ?G |- ?P Sometime ?RR'	sometime is monotonic
Mach_sometime_mplike: [| ?G |- Sometime ?RR; |- ?RR Implies (Sometime ?RR') |] ==> ?G |- Sometime ?RR'	structural rule for arguing with sometime
Mach_subst: [| ?a = ?b; ?H ?b |- ?P ?b |] ==> ?H ?a |- ?P ?a	if a and b are equal, substitute one for the other
Mach_thinL: [| LeftExtending ?H; $?H, $?G |- ?P |] ==> $?H, ?R, $?G |- ?P	drop an unnecessary hypothesis
Mach_thinR: $?H |- FalseV ==> $?H |- ?P	contradictory hypotheses can prove any goal
NotFirst_Previously_across_Not: |- Always ((Not First And Previously Not ?P) Implies Not (Previously ?P))	if not first can take not through previously
NotFirst_Previously_across_Not_rev: |- Always (Not First And Not (Previously ?P)) Implies Previously Not ?P	not through previously, other way

Rule	Explanation
fto_persists_also: |- Always (Previously (?RR And ?RR')) Implies ?RR' ==> (?P And ?RR') FromThenOnS ?RR |- (?P And ?RR') FromThenOnS (?RR And ?RR')	propagate verification through fromthenon

C.2 Rewriting equalities

The rewriting rules are either used as part of DOVE's simplification package or else repesent simplifications which it is sometimes useful to apply.

Rule
AndOr: (?p And (?q Or ?r)) = (?p And ?q Or ?p And ?r)
D99_AlwaysExp: (Always ?p) = (?p And Previously (Always ?p))
D99_FTOExp: (?p FromThenOn ?q) = (?q And (?p Or Previously (?p FromThenOn ?q)))
D99_InitFalse: (Initially FalseV) = FalseV
D99_InitTrue: (Initially TrueV) = TrueV
D99_NoAnd: (Not (?p And ?q)) = (Not ?p Or Not ?q)

Rule
D99_NoFalse: (Not FalseV) = TrueV
D99_NoOr: (Not (?p Or ?q)) = (Not ?p And Not ?q)
D99_NoSometime: (Not (Sometime ?p)) = (Always Not ?p)
D99_NoTrue: (Not TrueV) = FalseV
D99_SomeExp: (Sometime ?p) = (?p Or PreviouslyS (Sometime ?p))
D99_andState: (?P & ?Q) = (?P And ?Q)
D99_falseState: False = FalseV
D99_impState: (?P --> ?Q) = (?P Implies ?Q)
D99_noState: (~ ?P) = (Not ?P)

Rule
D99_nono: (Not Not ?P) = ?P
D99_orState: (?P | ?Q) = (?P Or ?Q)
D99_trueState: True = TrueV
FTO_And: (?P FromThenOn ?RR And ?RR') = ((?P FromThenOn ?RR) And (?P FromThenOn ?RR'))
Previously_across_And: (Previously (?A And ?A')) = (Previously ?A And Previously ?A')
Previously_across_ForAll: (Previously (ForAll x. ?A x)) = (ForAll x. Previously ?A x)
Previously_across_Implies: (Previously (?A Implies ?A')) = (Previously ?A Implies Previously ?A')
Previously_across_Or: (Previously (?A Or ?A')) = (Previously ?A Or Previously ?A')
conjTRUE2: (?P And TrueV) = ?P

Rule
conjTRUE1: (TrueV And ?P) = ?P
disjTRUE2: (?P Or TrueV) = TrueV
disjTRUE1: (TrueV Or ?P) = TrueV
impTRUE2: (?P Implies TrueV) = TrueV
impTRUE1: (TrueV Implies ?P) = ?P
conjFALSE2: (?P And FalseV) = FalseV
conjFALSE1: (FalseV And ?P) = FalseV
disjFALSE2: (?P Or FalseV) = ?P
disjFALSE1: (FalseV Or ?P) = ?P

Rule
ftos_def: (?p FromThenOnS ?r) = ((Sometime ?p) And (?p FromThenOn ?r))
impFALSE2: (?P Implies FalseV) = (Not ?P)
impFALSE1: (FalseV Implies ?P) = TrueV
prevs_def: (PreviouslyS ?p) = (Not (Previously Not ?p))

Appendix D State machine diagnostics

DOVE provides reasonably comprehensive diagnostics for syntactic errors in the definition of the state machine or its associated properties. These diagnostics may be generated either at the time of input or else during the operation of the Check/Compile option of the Edit menu.

Since all parsing is done directly in the Isabelle proof assistant, checks for the use of undefined variables, or the misuse of variables according to their declared type, etc, are immediately carried out upon input. In particular, simple spelling errors and convention clashes are usually very simply picked up this way.

D.1 Checks carried out by Check/Compile

The first check that is carried out determines whether or not all declarations (eg types, variables, transitions and so on) have been parsed, or committed. If not, a list of the offending declarations is presented to the user, and the user is responsible for seeing that each declaration is parsed correctly (by selecting the Commit button on the declaration's editor window, as discussed in detail below) before attempting to compile the state machine again. For this reason, it is in the interests of the user to ensure that most declarations, particularly basic building blocks such as types, variables, and so forth, are parsed/committed as they are declared.

The compilation process then proceeds to the syntactic stage. Syntactic checks are for

1. identifiers used but not declared, and
2. identifiers used twice, or which clash with those of the underlying DOVE tool.

These are fatal errors for the construction, and are written as such into the dialog box.

The structural checks are carried out to check that

- an initial state has been defined (in the Initialisation option of the Definitions menu).
- in the topology of the diagram every node on the state machine is reachable from the chosen initial state.
- no two nodes have the same name.

Any failure in these checks is a fatal error, and will be written as such into the dialog box.

If there have been no errors so far, the compilation process then proceeds to the next phase, in which DOVE proceeds to load the corresponding Isabelle theory file into the Isabelle proof tool which checks for syntactic errors. In particular, it carries out all the type-checking. This is a much more stringent test than the preliminary checking described above, and somewhat slower. The error messages which may possibly ensue are those of the Isabelle proof tool.

D.2 Diagnostic messages from Check/Compile

The error messages listed here are generated specifically for the DOVE tool. They will be output to a dialog box labelled Compilation -- Diagnostics. If the check succeeds, then the following message will appear

```
Success : No syntactic or structural errors found
```

If not, a number of the following may appear.

D.2.1 Initialisation checks

If there is a problem with the initialisation, under Syntax Check will appear the message

```
Error found when checking initialisation predicate:
```

followed by either:

```
No initial state has been defined (see Initialisation window).
```

if the initial state has not been entered in the Initialisation window; or,

```
The initial state name ‘‘stateName’’ is not a valid state name.
```

if statename is not one of the names assigned to the state machine nodes.

D.2.2 Edge to transition checks

A fatal error occurs if a given edge name is not declared as a transition,

```
Error : The edge ‘‘edgeName’’ has no corresponding transition.
```

However, a declared transition name not being assigned to an edge is not a fatal error. Still, it could be an oversight on the user's part, so DOVE gives a warning,

```
Warning : Some of the defined transitions
are not associated with edges:
```

listing the unassigned transition names.

D.2.3 Structure Checks

Problems with state machine structure are reported under `Structure Check`, producing:

```
Error : Missing initial state.
```

if the initial state has not been entered in the `Initialisation` window;

```
Error :  Repeated state names :
```

listing state names which are not uniquely assigned to nodes; or,

```
Error : Nodes unreachable from initial state :
```

listing nodes which cannot be reached by following the directed edges from the assigned initial state.

D.2.4 Uncommitted data

An error will be returned if the user attempts to compile whilst there are definitions which did not parse appropriately – so, the declaration window has been "Closed" instead of "Committed". For example, if the variable Var has failed to parse, the result of `Check/Compile` is

```
Compilation cannot be carried out, since the following
declarations have not yet been committed:

VARIABLES NOT COMMITTED:

       Var
```

D.3 Parsing errors in definitions

Since the input is directly parsed in the underlying Isabelle tool, any error messages are output straight from Isabelle and simply redirected to a diagnostic dialog box. As such, the meaning of the diagnostic will in general not be particularly clear! However, early alerting of the user to the fact that the error has occurred is expected to typically allow a quick correction, since at this input stage of the process the possible mistakes are quite limited. In this way, the user should also gain a feeling for which diagnostic is associated to which kind of error. In any case, we just restrict ourselves to giving a small subset of examples below of errors and the corresponding output diagnostic.

D.3.1 Type errors in defining type abbreviations and variables

Mis-spelling an Isabelle/HOL type, or previously defined type abbreviation, is a fatal error in general. The following results from declaring the type of a variable to be `Nat`, where `Nat` has not been defined as a type abbreviation (the Isabelle/HOL type of natural numbers is `nat`).

```
Invalid variable definition: Invalid variable definition -
Undeclared type constructor "Nat"
```

which is very easily understood.

Compare this to the error message which appears when instead trying to assign the function type `nat => nat`, and accidentally writing `nat ==> nat`.

```
Invalid variable definition: Invalid variable definition -
Inner syntax error at: "==> nat"
*** Expected tokens:  "EOF" "\<leadsto>" "~=>" "+" "\<times>" "*"
***   "\<Rightarrow>" "=>" "longid" "id"
*** The error(s) above occurred in type "nat ==> nat"

uncaught exception ERROR
  raised at: library.ML:1114.35-1114.40
             sign.ML:553.34
ML>
```

Note that the syntax error begins at the offending symbol, which is the most useful aspect of this output. The “Expected tokens” list can typically be ignored, and will be suppressed in the following.

D.3.2 Errors in transition input

If the key words of the transition definition (`Let:`, `Guard:`, and `Act:`) are not included, or even if the colon is omitted, then the input will not parse. The resulting error is

```
Invalid transition definition: Invalid text at start of transition.
```

Similarly, an error results if the incorrect assignment symbol is used

```
Invalid transition definition: Invalid transition definition -
Inner syntax error at: "<= 0 ;"
*** Expected tokens:  "<--"
*** The error(s) above occurred in axiom "ET_Tdef"

uncaught exception ERROR
  raised at: library.ML:1114.35-1114.40
             /home/jgm/DOVE_CVS/DOVE/src/isa/parse.ML:237.38
ML>
```

(the “Expected tokens” list is displayed since this example shows it can be useful when there are few alternatives); or the semicolon omitted,

```
Invalid transition definition: Invalid transition definition -
*** Inner syntax error: unexpected end of input
```

If the user assigns to assign to the same variable twice in a transition, the resulting error is

```
 Invalid transition definition: Invalid transition definition -
Error in parse translation for "@mk_trans"
"varName" assigned to multiple times

uncaught exception ERROR
  raised at: /home/jgm/DOVE_CVS/DOVE/src/isa/parse.ML:237.58-237.63
ML>
```

A more interesting error, perhaps, arises when the user attempts to assign a value of wrong type; here trying to assign `True` to a variable of type `nat`,

```
Invalid transition definition: Invalid transition definition -
*** Type unification failed: Clash of types "bool" and "nat".
*** Type error in application: Incompatible operand type.
***
*** Operator:
***   (TranTy.mk_action
***     (%x. (ConfDove.heap_Lval Parser.jv@loc (Values.vinj x)))) ::
***   [(??'a * ??'b) * ??'c * (Parser.heapDT => Values.VAL) => nat,
***   (??'a * ??'b) * ??'c * (Parser.heapDT => Values.VAL),
***   (??'a * ??'b) * ??'c * (Parser.heapDT => Values.VAL)]
***   => (??'a * ??'b) * ??'c * (Parser.heapDT => Values.VAL)
*** Operand:   (%c. True) ::
***   (??'a * ??'b) * ??'c * (Parser.heapDT => Values.VAL) => bool
***
```

Note that “unification” is the technical name for the pattern-matching done in the proof tool. Also, again the initial error message is useful, the remainder not so useful.

D.3.3 Errors in property input

The `Properties Manager` requires input which is a Boolean expression built from the variables, constants, constructors of the state machine theory and any Isabelle theory on which it is based. Moreover, a property can be stated containing free variables – not previously defined in the theory. Isabelle simply assumes them to be free, and assigns them the appropriate type to have an overall Boolean.

Thus, the (silly) expression `Always` leads to the following error,

```
Invalid property definition: Invalid rule definition -
*** Inner syntax error: unexpected end of input
```

since Isabelle parses and recognizes the syntax of the unary function `Always` which takes a Boolean argument and produces a Boolean. In contrast, the even more silly expression `always` is accepted by the parser, since isabelle simply treats it as a free variable and assigns it the type `bool`.

In fact, using the temporal operators in properties has a further subtlety. Even the "Boolean expression" `Always (1 = 1)` is no good, since there is no constant corresponding to `Always` defined in the theory – it is represented as pure syntax. The error which this produces is

```
Invalid property definition: Invalid rule definition -
*** No such constant: "@always"
```

Indeed, acceptable temporal properties in the state machine theory are structured, to streamline the user input. What is needed is the "Turnstile" operator to convert to a valid Boolean expression, such as

```
|- Always (1 = 1)
```

In contrast, properties defined which do not refer to temporal quantities, such as `(varName = 1)` (for some variable `varName` of type `nat`), do not need the Turnstile operator.

Errors from type clashes will give similar output to the corresponding errors seen earlier.

Appendix E Dealing with errors

The current version of the DOVE system, being a research prototype, contains a number of known flaws, and no doubt the dedicated user will find a few more. In this chapter a quick overview of the "error interaction" with Tcl/Tk is given, and then the known bugs or design flaws in each mode of the DOVE tool are listed.

E.1 General overview of error interaction

At various stages in the operation of the DOVE tool, the user will find that there are periods of enforced user inaction while the DOVE tool is busy performing lengthy computations. These periods are typically short, but may be up to tens of seconds when running on slower computers. Often, if the user looks at the background Isabelle terminal window it will be seen that the tool is far from inactive. In any case, it should happen at these times that a busy notice will appear indicating to the user that computations *are* being carried out in the background. When those computations are completed the busy notice should disappear.

If DOVE is loading, or is otherwise "busy", the user should wait until it has finished before continuing to enter commands via the mouse or keyboard. The platform is not completely robust as it stands, and can be "confused" by indiscriminant mouse button clicking while performing its computations. When confused, or when a bug has appeared, the tool will typically either freeze or will bring up a Tcl/Tk error dialog box.

In the former case, be sure that DOVE has "hung" and is not simply waiting for user input. If not then there is probably no option but to quit the current session (using the underlying process management system) and start again. This is not expected to be a common occurrence, but still the usual good practice of regular intermediate saving of ongoing work is recommended.

In the latter case, a Tcl/Tk error dialog box will appear over the DOVE state machine window, often together with an existing busy notice. Both of these boxes must be closed before the user may continue with the session. Sometimes the reported error will be trivial and it will be possible to continue the session without trouble, but it is a good idea to immediately save the design. However, the error may cause the session to "hang" and the user will then be forced to kill the session using the underlying process management system. Lack of response from the various DOVE windows is often due to a notice box which has not been closed rather than the system hanging. These notices are somewhat easy to overlook, in particular if the DOVE state machine window is iconified and hence the notice also!

On this point of grab and focus, it is worth noting that notice boxes can be moved if they are inconveniently placed while the user works on another application. However, if the mouse is in use during this time it is possible to be unlucky and clash with the grab of the Tcl/Tk interaction – which will then get confused and bring up an error dialog box. This usually simply requires repeating the DOVE operation which was in progress.

Finally, in this overview, it is worth noting the possibility that restarting the DOVE tool will not eliminate the error. The most probable cause of such behaviour is that the ML image file has become corrupted. Recall from Section 3.2 that for a state machine graph file `Machine.smg`, DOVE constructs an ML image file `Machine.MLextn` with extension determined by the ML compiler being used. This file is modified incrementally when, for example, changes are made or properties

are proved, and an error may make the ensuing modifications inconsistent. In the case of strange behaviour at startup, it is a good idea to delete this image file from the working directory and force DOVE to construct a new one. It will be necessary to select the `Compile` option in the `Edit` menu of the state machine graph window before any further proof or animation activities.

E.2 Known bugs or design flaws in DOVE

E.2.1 Edit mode

The user should be careful not to click the mouse button indiscriminantly when the DOVE tool is loading, or when a busy notice is displayed. The result *can* be, apparently randomly, that the Tcl/Tk interaction gets confused and the session freezes.

Some more specific problems with the interface are as follows.

- The `Initialisation` window will not come up again once quit. Thus, it should be edited but not closed until the DOVE session is concluded.
- The `Undo` facility does not work properly: if a node is deleted its attached edges are automatically also removed. Applying `Undo` then restores the node *without* the associated edges.
- It appears that a `Write Error` can occur unpredictably when trying to save the state machine – after sufficiently complicated editing (including deleting nodes and attached edges) is applied after storing an animation. The previous `machine.smg` file should *not* be overwritten.
- Property names should not include the word “error”, since this will give strange and subtle errors!
- Two properties cannot be given the same name, even if they are in different property sets.

E.2.2 Animation mode

There are, again, a few problems with the user interface.

- The `Animator` comes up with the “last” animation (stored). This is a “feature”, but the user should be careful to check the title bar to see what is actually in there!
- Following from the last item, if there is an animation loaded into the `Animator`, but the user wants to load an animation using the `Load File` option of the `File` menu, the animation should first be cleared via the `New` option.
- The `Watch Variables` window does not accept input properly, unless the user is careful to add the variable names *in order*. The problem is that the cursor preferentially goes to the right-hand pane even when clicking on the left-hand pane, if there exists an entry parallel to the point of click.

- Somewhat anti-intuitively, the `Description` frame of the `Animator` can be written to only via the dialog box which is invoked through the `Store` option of the `File` menu. It can be added to at any stage of the animation, the accumulated description is shown at all stages reached via the control buttons.

- Numerical input for variable values must be directly preceded by a `#`; e.g., `#3` instead of `3`.

- To have a permanent record of the stored animation (kept in the state machine file), the user must use the `Save` option under the `File` menu of the state machine graph window.

- The `Show Names`-type option of the `Prover` interaction does not currently exist in the animator, so the full-name syntax is not available. Since the transition guard syntax is very simple, this is not a huge problem.

E.2.3 Proof mode

There are a number of known problems with the XIsabelle interface, which are now outlined.

- There is a slight operational problem with the interface in the `Prover` window, due to subtleties of the Tcl/TK programming: to get the text insertion cursor to properly focus – for example, in inserting text for a `Basic Tactics` choice – it can be necessary to first move the mouse pointer off the `Prover` window and then return it to the desired position.

- The user should be careful when using the `Chop` options under the `Proof` menu of the `Prover` window. During a proof it can happen that the current subgoal after the "chop" is not the expected one. The user needs simply click on the desired subgoal as usual.

- A given line in the proof script of a `Proof History` window gives the tactic and the number of the subgoal to which it is applied in the proof. However, the subgoal number is not used by the XIsabelle interface when that line in the proof script is clicked on. XIsabelle will simply apply the corresponding tactic to the subgoal which is currently highlighted (the active subgoal). Thus the user must first click on whichever subgoal the tactic *should* be applied to, thus making it the active subgoal, before applying the tactic (double-clicking) from the `Proof History` window. For a similar reason, to carry out "Apply to All" the user should simply click once on the desired line of the proof script, and then press `Apply to All` on the `Prover` window.

- When multiple proof sessions are run simultaneously in the `Prover`, the proof scripts should be saved immediately upon completing the proof of each. If this is not done – say by first saving the proof of property "`prop2`" and then swapping back to save a previously completed proof of property "`prop1`" – Isabelle may get confused regarding the names of proofs. For example, in the `prop1.prf` file the last line of the proof script will read

  ```
  qed prop2
  ```

  instead of the correct `qed prop1`. If the user forgets to save at the appropriate time, this can easily be fixed by editing the file directly.

- There is a possible problem with the Matching On facility of the Theorem Browser window, called up when Match Theorems is pressed on the Prover. For some subgoals there may be ambiguities in the matching unification, and an error output will ensue. This may depend on whether the Show Brackets option – in the Options menu of the Theorem Browser window – has been toggled on or off, so some experimentation may remove the problem.
- If already opened, the Tactic Tree is *not* brought to the foreground when selected via the Tree Display option of the View menu.
- When there are a lot of subgoals, only the first few are displayed in full, and the rest are elided. An elided subgoal can be redisplayed in full by clicking on it with mouse button 2. However, this will not work for the last subgoal in the frame.
- The Term frame of the Theorem Browser window should not be edited by the user.
- The Help menu of the Theorem Browser window is malfunctioning.
- The Goal & Premises window – obtained via the Goal and Premises option of the View menu on the Prover – is not "clickable". At present, then, if the user has premises (unusual at least for temporal properties) in the goal these can only be included (as for usual interaction with Isabelle for an advanced user) by referring to the premise name as appearing in the Goal & Premises window.
- Selecting the Chop and Restart options of the Proof menu on the Prover will sometimes leave no current subgoal selected. The user should simply select the desired current subgoal.
- Note that to select any of the Interactive tactics of the DOVE Tactics appropriately the user must *double-click* (with mouse button 1).
- When entering a temporal property into the Current Tactic window in the application of an Interactive tactic, the user must currently prepend the theorem name with the name of the property set in which it appears. That is, a property Prop in property set Set is currently stored in the ML image as the theorem Set_Prop, and it must be invoked by this name.
- The Load Tactics option of the Tactic menu on the Prover is broken. Thus, user-defined tactics have to be redefined in each DOVE session.

Appendix F Troubleshooting DOVE

The following provides trouble-shooting information for problems which may arise in operating DOVE. The corresponding information for problems encountered when installing DOVE appears in the file `Install.ps` found in the distribution documentation.

Problem	Possible Cause	Solution
An error message `invalid spawn id` is displayed.	Isabelle did not start correctly as it did not have the ability to use the ML Compiler. This often happens if the compiler only has a licence for certain work stations.	Connect to or move your work to a work station which is able to use the ML Compiler
An error message indicates that `DISPLAY is set as :` with nothing following.	Your `DISPLAY` environment variable has not been set to anything at all.	Set the `DISPLAY` variable to your current work station. In `csh`, this is done using the `setenv` command.
DOVE is frozen.	It may be that DOVE has put up a modal dialog box that has become hidden.	Sort through the windows in your window manager until you find the dialog that needs a response.
An error message claims the Isabelle fonts cannot be found.	This is only a problem when attempting to use XIsabelle98.	This is a problem with the Isabelle setup on your system. Consult *The Isabelle System Manual* [12, Chapter 3]. Alternatively it is possible to use the font setting mechanism for your system.

DOVE cannot find an image file.	Due to the Isabelle image selection mechanism, logics not in the standard isabelle heaps directories must be specified with a relative path or an absolute path.	Use:
		`dove ./Image`
		or
		`dove $PWD/Image`

A prompt appears after starting the application, but no DOVE window.	Your ML prompt, which is specified in your settings file, is incorrect.	Check the contents of $HOME/isabelle/etc/settings. to ensure that it is set up for the correct version of ML.

Appendix G Questions and answers about the formal proof model

This "FAQ" appendix is intended to provide background for the interested reader. It may help to develop some degree of confidence in dealing with the `Prover`.

G.1 What is a formal theory, anyway?

To develop a formal theory for reasoning about a given system, the first step is to construct the *syntax* – namely, an alphabet and grammar. The rules of the grammar must be strict enough so that a statement in the corresponding *formal language* is unambiguous. The formal theory is then concretely specified as a set of *axioms* and *inference rules*. An axiom is a statement in the chosen *formal language* which is true a priori, without any further hypotheses – i.e., by definition of the theory. An inference rule is a purely syntactic rule for moving from one true statement to another. Here "truth" is simply definitional, meaning that the statement can be obtained from the axioms by application of inference rules. The statements which can be so-derived are called the *theorems* of the theory, and such a derivation of a given theorem is called its *proof*. Note that the axioms are theorems. This syntactic construction is often called a *deductive system*.

Any "meaning" attached to the deductive system is through *semantics*, which (loosely speaking) is an interpretation of the syntactic construction in terms of some familiar system. As an example, the reader will have seen the interpretation of the standard logical connectives in terms of Boolean truth tables. The interpretation will in general have an inherent notion of truth, so the question of whether a given statement holds for a given interpretation is simply whether it interprets to a true fact about the familiar system. A statement is semantically *satisfiable* if there exists an interpretation in which it is true. A statement is semantically a *consequence* of the axioms if for every interpretation in which the axioms are true, the statement is also true.

Thus, a formal language has a strict syntax and an unambiguous semantics, and the notions of truth can be compared. The deductive system is said to be:

- *sound* if all theorems are semantically a consequence of the axioms.
- *consistent* if there is no statement such that both it and its negation are theorems.
- *complete* if every statement which is semantically a consequence of the axioms is, in fact, a theorem.

Of these, the first is easily checked at the level of axioms and inference rules, and the second requires that the axioms are semantically satisfiable. The last is significantly more difficult.

Index

00-55, *see* assurance standards
00-56, *see* assurance standards

5679, *see* assurance standards

action, *see* transition, action
animation, *see* animator
animator, iii, 2, 7, 14, 24, 51–60
    importing, 56
    path condition, 52, 53, 59
    removing, 57
    selecting, 57
    storing, 56, 60
    watch variables, 52, 58
        setting values, 52, 59
assurance, *see* design, assurance
assurance standards, iii
attributes, *see* configuration
automatic proof tool, *see* proof, tool
axiom, *see* proof, axiom

behaviour, xv, 9

canvas, *see* graph, canvas
checked, *see* syntax checking
checking, *see* syntax checking
compiling, 32, 34, 40, 49, 51, 61
configuration, xv, 10–11, *see also* variable
    control, xvi, 10
    memory, xv
constant, 33, 35, 41, 47
control state, *see* configuration, control

deadlock, 20
declaration status, 34, 45, 62
definitions menu, *see* menu, definitions
design
    assurance, 2–3
    formal, xv
    modelling, iii, 2, 4
discharging a subgoal, *see* proof, subgoal

edge, 36, *see* graph, edge
edit menu, *see* menu, edit
editor, 6, 24, 27–50
execution, 13–14, 51, 80, *see also* configuration

Expect, 4

file
    image, 23, 31, 32
    noweb, 23
    PDF, 24, 31
    state machine graph, 23, 31
    theory, xvi, 23, 32
file menu, *see* menu, file
formal design, *see* design, formal
formal language, *see* logic, formal language
formal proof, *see* proof, formal

goal, *see* proof, goal
graph, 5, 6, 9, 27–31, 36, 42–44, 51, 54, 80
    canvas, 27, 28, 42
    edge, 27, 30–31, 42
    grid, 27, 28, 33, 37
    node, 27–29, 41, 42
graph editor, iii
grid, *see* graph, grid
guard, *see* transition, guard
GUI, *see* user interface

heap variable, *see* variable, heap
help menu, *see* menu, help
Higher Order Logic, *see* HOL
history, *see* execution
HOL, *see* Isabelle, HOL

image, *see* file, image
including theories, 37
inference rule, *see* proof, rules
initial predicate, *see* initialisation
initial state, *see* initialisation
initialisation, 12, 35, 41, 48
input variable, *see* variable, input
installation, 21
interactive proof tool, *see* proof, tool
invalid, *see* syntax checking
Isabelle, 4, 9, 75
    HOL, 9, 10
    image, *see* file, image
ITSEC, *see* assurance standards

lemma, *see* proof, theorem
liveness property, *see* logic, property

logic
formal language, xv
property, xvi, 14–19, 36
liveness, 19
safety, 19
proposition, xvi
sequent, xvi, 80
temporal, xvi, 14–17, 80
theory, xvi, 32, 71, 78
file, *see* file, theory
logic, higher order, *see* HOL
logic, semantics, 14

meaning, *see* logic, semantics
memory, *see* configuration, memory
menu
definitions, 33–37
edit, 32
file, 31–32, 55, 56, 60, 68–71
help, 37, 72
options, 63, 69, 72
proof, 68
swap, 69
view, 32–33, 69, 71
windows, 57
ML, 4
modelling, *see* design, modelling

node, *see* graph, node
noweb file, *see* file, noweb

options menu, *see* menu, options

PDF, 24, 37
Portable Display Format, *see* PDF
proof
axiom, xv
formal, xv
goal, xv, 79
rules, xv, xvi, 36, 75, 77
state, xv, xvi, 67, 79
strategy, 75, 80–85, 87–90
subgoal, xvi, 67, 79
discharging, xv
tactic, xvi, 67, 79, 85–87, 90, 91, 99–102
BackSubstitute, 80, 82, 83, 85, 89–90
ForwardsInduct, 80, 81, 85, 88
MasterBlast, 85, 96
Topology, 80, 82, 85, 88, 90, 91
tactics, 75
theorem, xv, xvi, 70, 71, 77
tool, 76
tools, xv, xvi
verification, 3, 17–19
proof menu, *see* menu, proof
property, *see* logic, property
property manager, 37, 61–66, 91
proving, 63, 66
saving, 63
sets, 63, 65, 91
status, 62
proposition, *see* logic, proposition
prover, iii, 4, 7, 24, 25, 63, 66–73, 75, 76, 79–81
display options, 69, 72
matching, 72
proof history, 69, 70, 79, 91
proof tree, 69
saving, 69, 70
swapping proofs, 69
theorem browser, 69–72
visualization, 72–73

safety property, *see* logic, property
saving, 31, 32, 34, 49
animator, *see* animator, storing
property manager, *see* property manager, storing
sequent, *see* logic, sequent
start state, *see* initialisation
state, 5, 9, *see also* configuration, control
state machine diagram, *see* graph
state machine graph, *see* graph
state machine graph file, *see* file, state machine graph
status, *see* declaration status
subgoal, *see* proof, subgoal
syntax checking, 34, 40–41
system attributes, *see* configuration
systems
critical, iii–iv, 1

tactic, *see* proof, tactic
Tcl/Tk, 4
temporal logic, *see* logic, temporal

temporal sequent, *see* logic, sequent
theorem, *see* proof, theorem
theory, *see* logic, theory
theory file, *see* file, theory
transition, xvi, 5, 9, 11–12, 27, 33, 36, 38–41, 48
    action, xv, xvi, 11, 39, 41
    guard, xv, xvi, 11, 38
    let, 11, 38
    renaming, 36, 49
truth tables, *see* logic, semantics
type, 10, 33, 35, 41, 44, 46

unchecked, *see* syntax checking
user interface, 3

variable, 5, 27, 33, 41, 47, 51, *see also* configuration
    heap, xv, 10, 12, 36, 41, 48
    input, xv, 10, 36, 41, 48
    names, 40
    watch, *see* animator, watch variables
verification, *see* proof, verification
view menu, *see* menu, view

watched variable, *see* animator, watch variables
windows menu, *see* menu, windows

XIsabelle, *see* prover

DISTRIBUTION LIST

Design Oriented Verification and Evaluation: The DOVE Project

Tony Cant, Brendan Mahony and Jim McCarthy

	Number of Copies
DEFENCE ORGANISATION	
Task Sponsor	
Defence Signals Directorate, Carolyn Dyke	4
S&T Program	
Chief Defence Scientist	1
FAS Science Policy	
AS Science Corporate Management	
Director General Science Policy Development	
Counsellor, Defence Science, London	Doc Data Sht
Counsellor, Defence Science, Washington	Doc Data Sht
Scientific Adviser to MRDC, Thailand	Doc Data Sht
Scientific Adviser Joint	1
Navy Scientific Adviser	Doc Data Sht
Scientific Adviser, Army	Doc Data Sht
Air Force Scientific Adviser	1
Director Trials	1
Information Sciences Laboratory	
Chief of Information Networks Division	1
Research Leader	Doc Data Sht
Head	1
Task Manager	1
Author	1
DSTO Library and Archives	
Library Edinburgh	2
Australian Archives	1
Capability Systems Staff	
Director General Maritime Development	Doc Data Sht
Director General Land Development	Doc Data Sht
Director General Aerospace Development	Doc Data Sht
Knowledge Staff	
Director General Command, Control, Communications and Computers	Doc Data Sht

Army

ABCA National Standardisation Officer, Land Warfare Development Sector, Puckapunyal	4
SO(Science), DJFHQ(L), Enoggera QLD	Doc Data Sht

Intelligence Program

DGSTA Defence Intelligence Organisation	1
Manager, Information Center, Defence Intelligence Organisation	1

Defence Libraries

Library Manager, DLS_Canberra	1
Library Manager, DLS_Sydney West	Doc Data Sht

UNIVERSITIES AND COLLEGES

Australian Defence Force Academy Library	1
Head of Aerospace and Mechanical Engineering, ADFA	1
Serials Section (M List), Deakin University Library, Geelong VIC	1
Hargrave Library, Monash University	Doc Data Sht
Librarian, Flinders University	1

OTHER ORGANISATIONS

National Library of Australia	1
NASA (Canberra)	1
AusInfo	1
State Library of South Australia	1

INTERNATIONAL DEFENCE INFORMATION CENTERS

US Defense Technical Information Center	2
UK Defence Research Information Center	2
Canada Defence Scientific Information Center	1
New Zealand Defence Information Center	1

ABSTRACTING AND INFORMATION ORGANISATIONS

Library, Chemical Abstracts Reference Service	1
Engineering Societies Library, US	1
Materials Information, Cambridge Science Abstracts, US	1
Documents Librarian, The Center for Research Libraries, US	1

INFORMATION EXCHANGE AGREEMENT PARTNERS

Acquisitions Unit, Science Reference and Information Service, UK	1

Library – Exchange Desk, National Institute of Standards and Technology, US	1
SPARES	
	5
Total number of copies:	47

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION DOCUMENT CONTROL DATA				1. CAVEAT/PRIVACY MARKING
2. TITLE Design Oriented Verification and Evaluation: The DOVE Project			3. SECURITY CLASSIFICATION Document (U) Title (U) Abstract (U)	
4. AUTHORS Tony Cant, Brendan Mahony and Jim McCarthy			5. CORPORATE AUTHOR Information Sciences Laboratory PO Box 1500 Edinburgh, South Australia, Australia 5111	
6a. DSTO NUMBER DSTO–TR–1349	6b. AR NUMBER AR 012–457		6c. TYPE OF REPORT Technical Report	7. DOCUMENT DATE October, 2002
8. FILE NUMBER	9. TASK NUMBER JTW 02/106	10. SPONSOR QR INFOSEC Branch, DSD	11. No OF PAGES 138	12. No OF REFS 14
13. DOWNGRADING / DELIMITING INSTRUCTIONS Not Applicable			14. RELEASE AUTHORITY Chief, Information Networks Division	
15. SECONDARY RELEASE STATEMENT OF THIS DOCUMENT *Approved For Public Release* OVERSEAS ENQUIRIES OUTSIDE STATED LIMITATIONS SHOULD BE REFERRED THROUGH DOCUMENT EXCHANGE, PO BOX 1500, EDINBURGH, SOUTH AUSTRALIA 5111				
16. DELIBERATE ANNOUNCEMENT No Limitations				
17. CITATION IN OTHER DOCUMENTS No Limitations				
18. DEFTEST DESCRIPTORS Modelling System Design			Critical Systems Verification	
19. ABSTRACT DOVE is a graphical tool for modelling and reasoning about state machine designs for critical systems. This report summarizes its technical development, and incorporates the user manual.				