Formal Modelling and Analysis of Broadcasting Embedded Control Systems

David Kendall

Ph.D. Thesis

September 2001

University of Newcastle upon Tyne
Department of Computing Science

In memory of William Kendall (1908 – 1994)

ABSTRACT

Embedded systems are *real-time, communicating* systems, and the effective modelling and analysis of these aspects of their behaviour is regarded as essential for acquiring confidence in their correct operation. In practice, it is important to minimise the burden of model construction and to automate the analysis, if possible. Among the most promising techniques for real-time systems are *reachability analysis* and *model-checking* of networks of *timed automata*. We identify two obstacles to the application of these techniques to a large class of distributed embedded systems: firstly, the language of timed automata is too low-level for straightforward model construction, and secondly, the synchronous, handshake communication mechanism of the timed automata model does not fit well with the asynchronous, broadcast mechanism employed in many distributed embedded systems. As a result, the task of model construction can be unduly onerous.

This dissertation proposes an expressive language for the construction of models of real-time, broadcasting control systems, and demonstrates how efficient analysis techniques can be applied to them.

The dissertation is concerned in particular with the Controller Area Network (CAN) protocol which is emerging as a *de facto* standard in the automotive industry. An abstract formal model of CAN is developed. This model is adopted as the communication primitive in a new language, *bCANDLE*, which includes value passing, broadcast communication, message priorities and explicit time. A high-level language, *CANDLE*, is introduced and its semantics defined by translation to *bCANDLE*. We show how realistic CAN systems can be described in *CANDLE* and how a timed transition model of a system can be extracted for analysis. Finally, it is shown how efficient methods of analysis, such as 'on-the-fly' and symbolic techniques, can be applied to these models. The dissertation contributes to the practical application of formal methods within the domain of broadcasting, embedded control systems.

ACKNOWLEDGEMENTS

I would like to express my gratitude to my supervisor Maciej Koutny for his patient support throughout the years it has taken me to produce this thesis. He has been extraordinarily generous in finding time for me, and his insistent probing of my ideas and rigorous attention to detail have helped me immensely.

Of course, I am indebted to the many researchers whose work is mentioned in this thesis, but I would like to acknowledge a personal debt to the following: Sergio Yovine and Stavros Tripakis for their help with KRONOS and OPEN-KRONOS, respectively; Hubert Garavel for support with CADP; and Gerard Holzmann and Jean-Charles Grégoire for sharing their code, from which I have learnt much about efficient implementation.

I have been fortunate to do this work surrounded by good friends and colleagues. Chris Phillips gave me vital support during my M.Sc. studies and has been a stalwart friend and adviser ever since. Everyone in the High Integrity Embedded Systems Group has provided friendly encouragement and contributed to a stimulating environment in which to do research. Steven Bradley, William Henderson and Adrian Robson have always been willing to listen to my ideas; they are responsible for thrashing out much of the chaff. Ljerka Beus-Dukic has never been short of an encouraging word, as I struggled through the final stages of 'writing up'. I consider myself particularly fortunate to have shared an office with William Henderson for more than a decade. He has been an unfailing source of good humour, good music, good advice and good friendship.

I would like to thank the School of Computing and Mathematics at the University of Northumbria, both for financial support and for the time which it has allowed me to devote to this research. In this regard, I am particularly grateful to Adrian Woolley for supporting my application for a secondment to get the work started, and for managing my teaching allocation sympathetically in subsequent years.

Above all others, I am grateful to Marilyn, my wife, best friend and greatest ally, without whose loving support I would have given up long ago, and to my children Caitlin, Martha and Josie, who always help to keep things in perspective, and whose love makes everything seem worthwhile.

PUBLISHED WORK

Preliminary versions of some of the work in this thesis have been presented at a number of conferences and workshops. Steven Bradley, William Henderson and Adrian Robson are co-authors of many of the following papers. The work presented in the thesis is entirely my own, except where explicitly acknowledged. The papers, in chronological order, are

- A formal basis for tool-supported simulation and verification of real-time CAN systems. In *Proceedings of 4th International CAN Conference (iCC'97)*, pages 719–727, Berlin, October 1997.
- *bCANDLE*: Formal modelling and analysis of CAN control systems. In *Proceedings of 4th IEEE Real Time Technology and Applications Symposium (RTAS'98)*, pages 171–177. IEEE Computer Society Press, June 1998.
- *CANDLE*: A high level language and development environment for high integrity CAN control systems. In *Proceedings of 4th IEE Workshop on Discrete Event Systems*, pages 58–63, August 1998.
- Using sharing trees in the automated analysis of real-time systems with data. In *Proceedings of IEE Colloquium: Applicable Modelling, Verification and Analysis Techniques for Real-Time Systems*, Ref. No.1999/006, pages 6/1-4. IEE, London, UK, January 1999.
- *CANDLE*: A tool for efficient analysis of CAN control systems. In *Proceedings of the 1st Workshop on Real-Time Tools (RT-TOOLS'2001), Aalborg, Denmark*, Technical Report 2001-014, University of Uppsala, August 2001.

My ideas concerning the translation from a process language to timed automata were developed first for the AORTA language. That work appears in

- Validation, verification and implementation of timed protocols using AORTA. In P. Dembinski, editor, *Proceedings of the Fifteenth International Symposium on Protocol Specification, Testing and Verification*, pages 205–220. Chapman and Hall, June 1995.

CONTENTS

1. Introduction 1
- 1.1 Embedded Control Systems 1
- 1.2 Formal Methods 3
- 1.3 Broadcast Communication 6
  - 1.3.1 Controller Area Network 6
- 1.4 The dissertation 8
  - 1.4.1 Justification 8
  - 1.4.2 Structure and contribution 9

2. Models, Specifications and Correctness 11
- 2.1 Introduction 11
- 2.2 Models of Time 11
- 2.3 Transition Systems 13
  - 2.3.1 Labelled Transition Systems 13
  - 2.3.2 Timed Transition Systems 15
  - 2.3.3 Composition of transition systems 16
- 2.4 Process Algebra 17
  - 2.4.1 Basic concepts 17
  - 2.4.2 Timed Extensions 18
- 2.5 Timed Automata 19
  - 2.5.1 Introduction 19
  - 2.5.2 Clocks 20
  - 2.5.3 Clock Constraints 20
  - 2.5.4 Syntax and informal semantics 21
  - 2.5.5 Formal Semantics 22
  - 2.5.6 Composition of timed automata 24
- 2.6 Property Specification 25
  - 2.6.1 State Properties 26
  - 2.6.2 Automata 27
  - 2.6.3 Temporal Logic 29
  - 2.6.4 Discussion 32
- 2.7 Verification 33
  - 2.7.1 Region Equivalence 33
  - 2.7.2 Region Graph 34
  - 2.7.3 Complexity of reachability 36
  - 2.7.4 Constraint Solving 38
  - 2.7.5 Difference Bound Matrices 43

2.7.6 Implementing constraint solving 48
2.7.7 Other attacks on state space explosion 52
2.7.8 Tools 56
2.8 Conclusions 57

3. *bCANDLE*: A low level modelling language 59
3.1 Introduction 59
3.2 Informal system model 59
3.3 The Data Model 61
3.3.1 Formal Definition 62
3.4 The Network Model 64
3.4.1 Structure 65
3.4.2 Behaviour 71
3.5 The Process Model 76
3.5.1 Syntax 77
3.5.2 Informal Semantics 79
3.6 Formal system model 81
3.6.1 Well-formed systems 81
3.6.2 Operational semantics 82
3.6.3 Strong equivalence 84
3.6.4 Equational laws 87
3.7 A simple example 88
3.8 Conclusions and Related Work 91
3.8.1 Broadcast communication and Real-Time 91
3.8.2 Process Operators 92

4. Analysis via Timed Automata 93
4.1 A *bCANDLE* System and its Timed Automaton 93
4.2 Models with explicit clocks 95
4.2.1 Clocked Networks 96
4.2.2 Clocked Process Terms 97
4.2.3 Safe Clock Allocations 100
4.2.4 Clocked *bCANDLE* systems 102
4.3 Timed Automaton Construction 102
4.3.1 Principles of construction 102
4.3.2 Construction of the automaton 103
4.3.3 Commentary on the construction 104
4.3.4 Correctness of the construction 107
4.4 Implementation of the construction 108
4.4.1 Nets 109
4.4.2 Constructing the net for a clocked term 112
4.4.3 Final stage of timed automaton construction 121
4.5 A simple example 122
4.6 Conclusions 125

5. Space-Efficient, On-the-fly Reachability Analysis 126
5.1 Introduction 126
5.2 On-the-fly reachability analysis 127
5.2.1 Basic algorithm 127
5.2.2 Clock activity reduction 128
5.3 A Minimised Automaton Representation of Reachable States . . 132
5.3.1 Minimised Deterministic Finite State Automata 133
5.4 Implementing a MA state store for *bCANDLE* 135
5.4.1 The state vector 135
5.4.2 Mapping the state vector to MA layers 137
5.4.3 Variable Ordering 138
5.5 An experimental platform 139
5.5.1 The *bCANDLE* Compiler 139
5.5.2 State Space Storage Modes 139
5.6 Experiments 141
5.6.1 System models 141
5.6.2 Experimental results 141
5.6.3 Discussion of experimental results 142
5.7 Related work 143
5.8 Conclusions and further work 145

6. *CANDLE*: Modelling and Analysis in Practice 146
6.1 Introduction 146
6.2 A Tour of *CANDLE* 146
6.2.1 Modules 147
6.2.2 Data declarations 148
6.2.3 Expressions 151
6.2.4 Statements 152
6.3 *SDML*: Simple Data Modelling Language 159
6.3.1 Types 160
6.3.2 Constants 161
6.3.3 Expressions 161
6.3.4 Functions and Procedures 161
6.3.5 Statements 162
6.3.6 Semantics 163
6.4 Constructing a Formal Model 164
6.4.1 Declarations 165
6.4.2 Behaviour 167
6.4.3 An example 172
6.5 The *CANDLE* Development Environment 175
6.5.1 Overview 175
6.5.2 Validation Environment 177
6.5.3 The OPEN/CÆSAR Architecture 178
6.5.4 Model Generation 178
6.5.5 Model Exploration 179
6.6 An example 180
6.6.1 The *CANDLE* program 180

6.6.2 The *bCANDLE* model 182
6.6.3 Analysis of the model 183
6.7 Conclusions and Related Work 187
6.7.1 Conclusions 187
6.7.2 Related Work 187

7. Conclusions and Further Work 189
7.1 Conclusions 189
7.2 Further Work 189

Appendix 191

A. Flow Regulator TA 192
A.1 KRONOS `.tg` Format 192
A.2 Flow Regulator TA 192

B. Proofs 195
B.1 Correctness of the translation 195

C. The *CANDLE* Grammar 209
C.1 Syntax Notation 209
C.2 Lexical Conventions 210
C.3 Modules 210
C.4 Declarations 210
C.4.1 Type Declarations 211
C.4.2 Constant Declarations 211
C.4.3 Variable Declarations 211
C.4.4 Function and Procedure Declarations 211
C.4.5 Channel Declarations 212
C.4.6 Exception Declarations 212
C.5 Expressions 212
C.6 Behaviour 213
C.6.1 Send statement 214
C.6.2 Receive statement 214
C.6.3 Elapse statement 214
C.6.4 Assignment statement and Procedure Call 214
C.6.5 If statement 214
C.6.6 Repetition statements 214
C.6.7 Select statement 215
C.6.8 Trap statement 215
C.6.9 Module Instantiation 215

D. The *SDML* Grammar 216
D.1 Introduction 216
D.2 Data Modules 216
D.3 Declarations 216
D.3.1 Type Declarations 216

D.3.2 Constant Declarations 217
D.3.3 Function and Procedure Declarations 217
D.3.4 Variable Declarations 218
D.4 Expressions 218
D.5 Statements 219
D.5.1 Assignment statement and Procedure Call 219
D.5.2 Return statement 219
D.5.3 If statement and Repetition 220

E. Glossary 221

Bibliography 226

LIST OF FIGURES

1.1 Simple Embedded Control System . . . 2
1.2 CAN Frame – Standard Format . . . 8

2.1 A simple timed automaton . . . 22
2.2 Product construction for timed automata . . . 24
2.3 Level Crossing Control System . . . 25
2.4 `Test` automaton for bounded response . . . 28
2.5 TBA for bounded response . . . 29
2.6 Clock regions on $\{h_1, h_2\}$ with $c_1 = c_2 = 2$ . . . 35
2.7 Region graph reachability . . . 37
2.8 Convex and Non-convex Polyhedra . . . 41
2.9 Operations on Polyhedra . . . 42
2.10 Representation of a convex polyhedron by DBM's . . . 44
2.11 Weighted graph interpretation of a DBM . . . 45
2.12 Procedure to compute the canonical form of a DBM . . . 46
2.13 Convex decompositions of a non-convex polyhedron . . . 48
2.14 An algorithm for reachability based on the simulation graph . . . 51

3.1 Control system model . . . 60
3.2 Transmission Status Notation ($m \in M$ and $t_1, t_2 \in \mathbb{R}_\infty$) . . . 68
3.3 Rules for Network Behaviour . . . 75
3.4 Example of network behaviour . . . 76
3.5 Rules for Basic Systems . . . 83
3.6 Rules for Guard, Sequential Composition, Choice and Recursion . . . 84
3.7 Rules for Interrupt and Parallel Composition . . . 85
3.8 Flow regulator in *bCANDLE* . . . 89
3.9 Simulator trace of the flow regulator example . . . 90

4.1 One-shot flow regulator in *bCANDLE* . . . 94
4.2 A timed automaton for the one-shot flow regulator . . . 95
4.3 Rules for Network Edges . . . 104
4.4 Rules for Basic System Edges . . . 105
4.5 Rules for Guard, Sequential Composition, Choice and Recursion Edges . . . 106
4.6 Rules for Interrupt and Parallel Composition Edges . . . 107
4.7 Invariant function $I : b\widehat{CAN} \rightarrow \Psi_{\mathcal{H}}$ . . . 108
4.8 Example Net . . . 110
4.9 Rules for fire . . . 113
4.10 Net for a basic term . . . 113

4.11 Net for a sequential composition 114
4.12 Net for a data-guarded term 114
4.13 Net for a choice 115
4.14 Net for an interrupt 116
4.15 Net for a recursion 118
4.16 Compact net for a recursion 118
4.17 A recursion with indirections 119
4.18 A recursion with indirections removed 120
4.19 Algorithm to remove indirections 120
4.20 Algorithm to construct a timed automaton 122
4.21 The flow regulator revisited 123
4.22 Net for the flow regulator 123

5.1 Algorithm for on-the-fly reachability for *bCANDLE* 128
5.2 A minimised automaton 134
5.3 Structure of a *bCANDLE* state vector 135
5.4 Simple DBMs 136
5.5 State vector representation of a 3-clock zone 137
5.6 Orderings of the cells of DBM M'' (see Figure 5.4) 139

6.1 Flow Regulator: Instantiated and Renamed 159
6.2 Flow Regulator in *CANDLE* 173
6.3 *CANDLE* Development Environment: Architecture 176
6.4 *CANDLE* Validation Environment: Architecture 177
6.5 The Steam Boiler module 180
6.6 Water-level Sensor and Pump modules 181
6.7 Controller module 182
6.8 Steam Boiler Data Module 183
6.9 A *bCANDLE* model for a simple boiler controller 184

LIST OF TABLES

3.1 Example of Transmission Latency Functions 67
3.2 Equational laws 87

5.1 Test systems 141
5.2 Comparison of storage modes 142
5.3 Impact of variable ordering on minimised automaton modes . . 142

1. INTRODUCTION

This dissertation is concerned with the formal modelling and analysis of embedded control systems. We adopt the view that the construction and analysis of a formal model can contribute significantly to increased confidence in correct system operation. Attention is directed to distributed systems whose components communicate using a broadcast communication network. The deployment of such systems is becoming increasingly common, and ensuring the reliable fulfilment of their intended function is a challenging problem. In the rest of this chapter, the topics of embedded systems, formal methods and broadcast communication are introduced. The chapter concludes with a review of the approach and contribution of the dissertation.

1.1 Embedded Control Systems

Embedded computer systems [Kop97] are pervasive in the electronic equipment upon which we all are coming to depend. Applications range from household products such as microwave ovens, video recorders and cellular phones to control systems for the transportation, chemical, electrical, gas, oil and nuclear industries. What these computer systems have in common is that they are *embedded in a physical environment* with which they are required to interact for the purpose of control or monitoring. The role of the computer system in such interaction is typically

- to monitor significant variables of the environment such as temperature, pressure, flow or level;
- to execute a control algorithm which takes as its input the values of environmental variables and compute output values in accordance with one or more mathematical models of the physical system;
- to use values computed by the control algorithm to generate signals to the environment in order to control its function or optimise its performance.

The function of monitoring the environment is performed by physical sensors within it. For example, a thermocouple produces an analogue signal (a voltage) which varies with the temperature of the environment in which it is placed. A digital value is obtained from an analogue signal by A/D conversion, calibration and transformation to standard measurement units (e.g. degrees Celsius) in a process known as *signal conditioning*. Such digital values are the inputs to the control algorithms of the computer system.

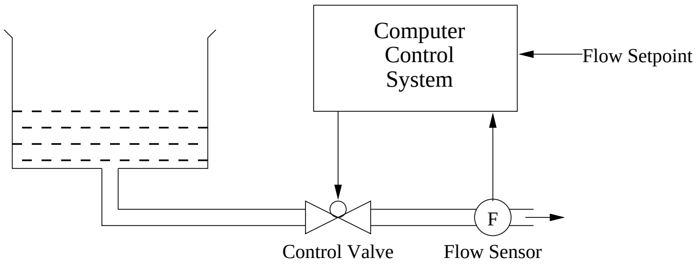


Fig. 1.1: Simple Embedded Control System

Control algorithms are developed by control engineers who understand the behaviour of the physical environment. The function of a control algorithm is to generate output signals to the environment to influence its behaviour so that some performance criterion is satisfied, even in the presence of random disturbances.

Output from control algorithms is transmitted to the environment in digital or analogue form. For example, a digital output may cause a heating element to be turned on or a valve to be closed, or an analogue output, generated by a D/A converter, may vary a demand voltage to an electric motor in order to control its speed.

Figure 1.1 illustrates a simple embedded control system [Kop97]. The objective of the control system is to maintain the flow of liquid through a pipe at a set rate, despite changing environmental conditions: varying level of liquid in the vessel or temperature sensitive viscosity of the liquid, for example. The computer interacts with its physical environment by monitoring the rate of flow, using the flow sensor F, and adjusting the position of the control valve to bring the flow rate as close as possible to the set-point.

In many systems, control is *distributed* among several computing nodes interconnected by a communication network [Tör98]. A distributed computing system architecture is often a 'good fit' with the distributed nature of the physical environment. Cooperating control units can be placed close to the physical devices which they control, communicating with each other via a simple computer network rather than the expensive and heavy wiring harness of traditional control systems. A distributed architecture also accords with sound design principles such as modularity, dependability and scalability [Kop97].

The emphasis in this work is on techniques for increasing confidence in aspects of distributed system *dependability*. Laprie [Lap90] identifies dependability as being concerned with those attributes of a computer system pertaining to the quality of service which it delivers to its users over an extended period of time. It is clear that failure of an embedded system to deliver an acceptable quality of service may have catastrophic consequences, either for the safety of the physical environment or for the economic soundness of the system's supplier, which may suffer as a result of the need to recall or repair many faulty units of a mass produced commodity.

A crucial aspect of the dependability of an embedded system is its ability to react to stimuli from the environment in a timely way. More precisely, an embedded control system is a *real-time* system whose correctness depends not only on the logical results of computations, but also on the physical instants at which those results are produced [Sta88]. Real-time systems are classified as either *hard* or *soft*. A *hard real-time system* is a real-time system in which a single failure to produce a correct result within a specified interval of time is regarded as unacceptable. A *soft real-time system* is one in which a (usually small) number of such failures, over a given period of time, can be tolerated.

In this dissertation, we treat hard real-time systems, and are particularly concerned with techniques which seek to contribute to the assurance of system dependability by demonstrating that temporal requirements are satisfied under all possible workloads. Such techniques rely upon the *predictability* of the temporal properties of all aspects of system behaviour, including worst case execution times of application code and operating system services, and also communication latencies and hardware performance [CVGH98, Hal93, HS91]. The requirement for predictability demands simplicity in system design, and when necessary, flexibility and resource utilisation are sacrificed by adopting *static structures* which can be analysed at design time.

1.2 Formal Methods

Formal methods entail the use of mathematically based languages, techniques and tools for developing and reasoning about computer hardware and software. The mathematics required is usually *discrete mathematics*, incorporating ideas from set theory and logic. The use of mathematics has an impact both on the *descriptive* and on the *analytical* tasks which are required in the development of a computer system. For example, a descriptive task, such as specifying a set of requirements, can be accomplished precisely, concisely, and unambiguously using a mathematical language. Similarly, an analytical task, such as demonstrating that a program function correctly implements a high-level design, can be discharged convincingly using a mathematically rigorous argument. The objectives in applying a formal method are to achieve clarity and precision in description, and to reduce reliance on human intuition and judgement in analysis, making greater use of mathematical *calculation*.

This broad framework allows for a variety of *levels of formality* in the application of formal methods within a project. The NASA guidebook [Nat97] identifies the following:

1. Level 1 methods involve the use of notations and concepts derived from discrete mathematics in order to develop more precise requirements statements. Analysis, if any, is informal. There are no mechanical tools (computerised algorithms) to support the writing or analysis of formal expressions.

2. Level 2 methods involve the use of formalised specification languages with mechanised support tools ranging from syntax checkers and pretty-printers to type checkers, interpreters and animators. Usually, tool support for eliciting or checking mathematical arguments is not available.

3. Level 3 methods involve the use of formal languages with rigorous semantics and correspondingly formal methods of semantical analysis which support mechanisation.

Wolper [Wol97] categorises methods at levels 1 and 2 as 'weak' formal methods and methods at level 3 as 'strong'. His opinion is that

> Without semantical analysis formal methods are of limited value with respect to their stated goal of ensuring the correctness of software systems: their formal syntax and semantics are just theoretical properties, not assets that are exploited in a substantial way. From the point of view of the author, a strong formal method even with limited applicability is more meaningful than a weak one that is perfectly general.

There is a similar latitude in the *scope of application* of formal methods within a project. For example, some stages in the development life cycle may be singled out for particular attention, certain system components may be identified as critical to mission success or safety, and some system properties may be judged particularly important and worthy of special attention.

Careful decisions are needed about the appropriate level of formality and scope of application for each individual project, so that a good balance can be achieved between the costs and benefits of formalisation.

In this work, we consider the problem of constructing formal models of distributed embedded control systems, and of providing mechanical assistance for the analysis of their functional and temporal properties. So the focus is on 'strong' formal methods, in Wolper's sense. As to scope of application, it is often acknowledged that a formal model is useful in the design stages of a computer system, as it facilitates the early detection of bugs and helps to avoid expensive implementation of a faulty design. This is certainly the case. In addition, however, we wish to emphasise the usefulness, for embedded systems particularly, of a formal model of (some features of) the implementation. The satisfaction of temporal properties of the system usually depends crucially on implementation decisions whose details may not be available in the early stages of design. For example, choice of processors and communication mechanisms, task and message allocation and priorities, scheduling policies, and so on, can all have a significant effect on real-time performance. It is important to take steps to gain assurance that temporal requirements which are satisfied by the design are also preserved in the implementation.

Experience suggests that successful application of formal methods in an industrial setting depends upon a number of factors, including

- the use of expressive languages which are accessible to system designers, being 'intuitive' and 'easy to use';
- the availability of computer-based tools which provide prompt and useful feedback to their users; and

- the ability to integrate the formal method into a familiar development methodology, so that the method augments, rather than replaces, traditional techniques.

Many prominent formal methods are very expressive within a given context: e.g., Z [Spi88] and VDM [Jon90] offer the full generality of set theory and predicate calculus; Petri nets [Mur89] offer a general model of concurrency, and Hybrid Automata [Hen96] allow the expression of a wide variety of timed systems. However, there is a growing interest in *domain specific* languages [JW96], which sacrifice generality of expression in order to offer the system designer a more familiar syntax, a greater ease of expression for typical applications within their domain, and the possibility of tractable analysis supported by software tools. It is hoped that these advantages can weaken resistance to the application of formal methods in industry by reducing the cost of model building and analysis. This is the approach followed here.

The need for automation and the provision of useful feedback to the user has led to the increasing popularity of a style of analysis known as *model checking* [CGP99]. Model checking is a technique which relies on building a finite state transition model of a system and checking that a desired property holds in that model. The basic procedure in model checking is exhaustive state space search, which is guaranteed to terminate since the model is finite. Once the model has been constructed and the property of interest specified, the checking is entirely automatic. Furthermore, in the case that the property does not hold in the model, a counterexample is generated, which can provide the designer with valuable insight into the behaviour of the system and aid in debugging.

The main obstacle in the application of model checking to industrial scale systems is the size of the state spaces which arise in exhaustive search. This is known as the *state explosion problem.* There are many techniques for attacking this problem (§2.7.7). Here, we mention the importance of *abstraction* [LGS$^+$95]. An abstract model omits detail from the system description. However, it retains sufficient detail to preserve system properties of interest. In this way, the size of the state space to be searched is reduced and useful questions can be decided in practice. Some abstract models are *exact*, i.e., for all properties of interest, the property holds for the model iff it holds for the system. Other abstract models are *conservative approximations*, i.e., if a property holds for the model, it also holds for the system; but, if it does not hold for the model, its status with respect to the system is undecided. Clearly, an exact abstraction is desirable, but a conservative approximation may lead to a greater reduction in the size of the state space. If a property fails to hold in a conservative approximation, further investigation is required to determine if the failure is a genuine feature of the system, or an aberration caused by the approximation. Conservative approximations have been used successfully to analyse the behaviour of embedded system implementations [BHKR94, Cor96] and they are used extensively in the rest of the dissertation.

Even an abstract formal model can produce a state space which is too large to search completely in a reasonable amount of time or memory. Nevertheless, the model can be used effectively for *debugging* the design or implementation from which it is derived. The focus here changes from verification based on exhaustive search to *falsification* based on semi-exhaustive search [FKFV99]. Techniques motivated by this point of view include state storage methods which allow a small probability that some reachable states are not explored [Hol95, Ste97] and simulation techniques which aim for a saturated coverage of the state space [GA98, YSAA97]. The coverage provided by these methods can improve significantly on traditional validation techniques such as simulation and testing [Mye79].

In summary, formal methods are one important approach among several for gaining increased confidence in system dependability. The benefits include increased understanding gained from the construction and analysis of formal models, improved communication made possible by formal documentation, and a formal basis provided for the construction of software tools to assist in system development.

1.3 Broadcast Communication

The communication architecture encountered most frequently in the implementation of distributed embedded control systems is the *broadcast bus* [UK94]. In broadcast communication, a message transmitted from a single sending node can be received directly by all nodes connected to the network. This contrasts with point-to-point communication in which messages are transmitted from a single sender to a single receiver. The use of a broadcast bus simplifies implementation of the common requirement in an embedded system to provide a consistent view of the state of the physical environment to a number of different nodes, e.g., to a man-machine interface, a process control node and an alarm-monitoring node [Kop97]. It can also simplify the implementation of clock synchronisation and the tolerance of individual node failures.

A wide variety of broadcast protocols is seen in practice, each offering a solution to the problems posed by a particular application area, e.g., Profibus [DIN89] for process control, LON [Ech91] for building automation and CAN [ISO92], TTP [KG93] and QWIK [JO99] for automotive applications. It is not our intention to review this extensive field here. Surveys of the relevant principles and applications can be found in [Kop97, KS97, UK94, Ver97b]. However, we do offer a more detailed consideration of one such protocol, CAN, which is the basis of the formal model presented later in the dissertation and will serve as our canonical example of broadcast communication.

1.3.1 Controller Area Network

Controller Area Network (CAN) [Bos91, ISO92] is a simple, deterministic, broadcast communication protocol which is not only attractive to system developers but also amenable to formal modelling and analysis. It is gaining increasing importance and attention in the implementation of distributed real-time systems, as evidenced by the variety of contributions in the proceedings of recent International CAN Conferences [CiA99].

CAN provides multi-master, priority-based bus access using a CSMA/CD protocol similar to Ethernet's, but with a deterministic collision resolution policy which makes it suitable for use in hard real-time systems. It is a robust protocol offering high reliability even in harsh electromagnetic environments and is suitable for the transmission of short messages over a small area at speeds of up to 1 Mbit/s. CAN was developed by Bosch in the mid-eighties in order to reduce the need for complex wiring harnesses in the automotive industry. Its use in the European car industry has grown to the point where it is an acknowledged industry standard and its popularity is growing in the USA where it has been accepted as a standard by the SAE for bus and truck manufacture [SAE92]. The availability of low cost components from a variety of manufacturers, who are seeking to satisfy the high volume requirements of the automotive industry, has encouraged the use of CAN in an expanding range of application areas, including: medical, packaging control, agricultural machinery, lift control, measurement, robot control and PLC controlled manufacturing.

CAN Operation

Information is transmitted as fixed format *frames* which consist of a *message identifier*, 0 to 8 *data bytes* and sundry *control* bits as shown in figure 1.2. The physical medium is usually twisted pair cable over which frames are transmitted using NRZ encoding with stuff bits inserted when needed to preserve synchronisation. When the bus is idle, any connected node may start to transmit a new frame. If two or more nodes start to transmit frames at the same time, the bus access conflict is resolved by non-destructive bitwise arbitration which is based upon the message identifier. The bitwise arbitration mechanism classifies bits as either *dominant* or *recessive*. During transmission of the arbitration field, transmitting nodes monitor the bus. Transmission of a dominant bit by any node causes all nodes to monitor a dominant bit on the bus; only if all transmitting nodes send a recessive bit is the monitored bit recessive. If the transmitted bit is recessive, but a dominant bit is detected on the bus, the transmitting node recognises that it has lost the bus arbitration, ceases transmission of its frame and behaves as a receiver of the highest priority competing frame. In a standard CAN frame, the arbitration field consists of the message identifier and the RTR (Remote-Transmit-Request) bit. A message identifier consists of 11 (29) bits in the standard (extended) frame format and is interpreted as a non-negative integer assigning a priority to the frame. Priorities are assigned in monotonically decreasing order starting from 0. The transmitter with the frame of highest priority gains bus access *without* experiencing any delay due to the access conflict, i.e. it behaves as if it were the only node seeking access to the bus. This property makes the bus particularly suitable for predictable, real-time communication. Frames which are disturbed either by losing arbitration or by the occurrence of errors during transmission are retransmitted automatically when the bus becomes idle again. A frame which is retransmitted is handled like any other frame, i.e. it participates again in the arbitration process in order to gain bus access.

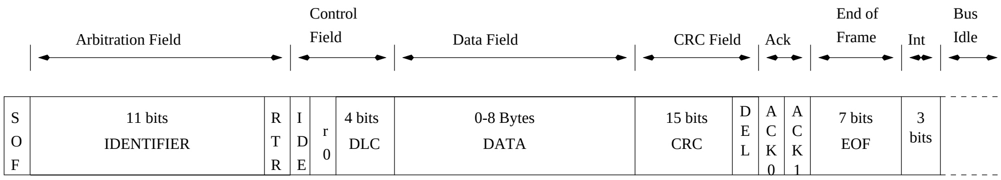


Fig. 1.2: CAN Frame – Standard Format

In addition to giving a priority to a frame, the message identifier is also used by each receiving node to determine whether or not it wishes to 'accept' the frame. There is no address associated with a frame to indicate its intended recipient. Each node connected to the bus performs an acceptance test during, or shortly after, the transmission of a frame. If the frame passes the test, its data field is made available to the accepting node, otherwise the node ignores the frame.

CAN-based protocols and analysis

There has been much interest in developing CAN-based protocols and analysis to solve a variety of typical distributed system problems. Tindell et al. [THW94] show how fixed priority pre-emptive scheduling analysis can be applied in order to bound message response time for systems with a suitably restricted computational model [TBW95]. Another approach to message scheduling is presented in [LKJ99], in which hard real-time messages are allocated off-line to slots in a Time Division Multiple Access (TDMA) schedule [KS97], with redundant time slots provided to achieve some fault tolerance; the redundant slots are used in the Earliest Deadline First (EDF) scheduling [KS97] of soft real-time messages, in the case of error-free transmission. Veríssimo et al. [VRM97] derive bounds for bus inaccessibility under a variety of fault scenarios. Protocols for achieving atomic broadcast in the presence of network faults are given in [RVA$^+$98] and [LK99]. A solution to the problem of fault-tolerant clock synchronisation is presented in [RGR98]. The only other formal study of CAN-based communication, so far as we know, is the Z specification of the protocol by Benzekri and Bruel [BB97]; however, real-time and performance aspects are not discussed in their work.

1.4 The dissertation

1.4.1 Justification

The work presented in this dissertation addresses the problem of providing a *high-level language* for modelling embedded systems which communicate using *broadcast communication*, with a view to exploiting *efficient, automated analysis techniques* in order to increase confidence in the satisfaction of temporal system properties. We briefly justify our belief in the need for work in these areas.

High-level language We have argued that a formal approach to system development is an important component in the construction of dependable systems, and that high-level languages and computer-aided analysis are required if formal methods are to be of practical use in industry. Much recent research on formal analysis of real-time systems has concentrated on techniques based on timed automata [AD90]. However, the language of timed automata is generally acknowledged to be too low-level for general use [AD94, BFK$^+$98, Tri98, Pet99]. Therefore, there is a need for research on methods to exploit the analysis techniques developed for timed automata in the context of high-level languages for modelling and development.

Broadcast communication An increasing number of distributed embedded systems are implemented which rely on broadcast communication. CAN is a simple, predictable broadcast protocol which is coming to dominate a large sector of this market. As it is often employed in systems which demand high dependability, there is considerable interest in the question of how to apply formal methods in the development of CAN-based systems. Currently available methods, however, do not offer a straightforward way to model systems which communicate via the CAN protocol. Our work is aimed at providing such a method.

Efficient analysis A high-level language for modelling broadcast systems will only be useful in so far as there are efficient techniques for analysing the models which are described with it. Our work shows how existing techniques can be applied and also proposes new techniques for efficient analysis.

1.4.2 Structure and contribution

Chapter 2 introduces labelled timed transition systems as a basic model for real-time systems and describes how such models can be derived using either timed automata or timed process algebra. The use of automata and temporal logic for the specification of system properties is presented. Techniques for verifying that a timed model possesses specified properties are described in some detail. This chapter presents no new results but is the foundation on which the rest of the dissertation is built.

Chapter 3 presents a new system modelling language, called *bCANDLE*, which allows the expression of process behaviour using a small set of process operators, includes primitives for broadcast communication based on a CAN-style protocol, and permits the modelling of both data and control structures. It is shown that the language satisfies a number of algebraic laws and is expressive enough to model essential features of CAN communication, including message priorities and channel latency, as well as standard real-time constructs, such as timeouts and watchdog timers. So far as we know, this is the first formally defined language which treats broadcast communication with prioritised message passing over latent channels in a dense time framework.

Chapter 4 defines a translation to timed automata for a large subset of *bCANDLE* systems. An efficient method for performing the translation is described and implemented. This work builds upon and extends the approach developed by Garavel in the translation of LOTOS [Gar92] and of Yovine in the translation of ATP [Yov93]. We demonstrate the use of the method by applying it to a simple *bCANDLE* model which is analysed using the KRONOS [BDM+98] model-checking tool.

Chapter 5 presents two techniques for efficient analysis of *bCANDLE* models: firstly, an on-the-fly generation of the simulation graph, incorporating clock activity reduction; secondly, a BDD-like, compact representation of the state space which treats discrete data variables and clock variables in a uniform manner. The application of the latter technique to timed systems is entirely novel. The former technique is based upon a combination of methods which is presented here for the first time.

Chapter 6 serves to validate the ideas presented in Chapters 3–5, and to point the way to future developments. It presents a high-level modelling language whose semantics are given by translation to *bCANDLE*, so providing a route to the use of the numerous analysis techniques based on timed automata, including those introduced in Chapter 5. The framework of a practical modelling and analysis environment is outlined. The utility and limitations of the techniques are illustrated in a small case study.

Chapter 7 summarises the work and suggests lines of future enquiry. Related work is referred to and discussed in context.

2. MODELS, SPECIFICATIONS AND CORRECTNESS

2.1 Introduction

Very simply, the use of formal methods in the development of a computing system involves:

1. the construction of a symbolic representation of (part of) the system, which captures what are believed to be essential features of its structure or behaviour. We call this symbolic representation *a model*.

2. the construction of a symbolic representation of some desired property of the system's structure or behaviour. We call this symbolic representation *a specification*.

3. the demonstration that the property described by a specification is exhibited by a model of the system. Such a demonstration is said to establish the *correctness* of the model with respect to its specification.

There is a wide variety of languages for expressing models and specifications, and of methods for establishing correctness. In this chapter, we introduce in some detail those languages and methods which are relied upon later in the dissertation. We also give a brief review of alternatives.

Most models of real-time systems, and specifications of their properties, employ a representation of Time. The representation which we use is introduced in §2.2. In §2.3, we introduce *labelled transition systems* and their *executions*, which serve as a unifying model of computation for both system models and specifications. Labelled transition systems can be described using several languages, including *process algebra* and *automata* which are the topics of §2.4 and §2.5, respectively. Specifications can also be given as automata, but in addition we use *temporal logic*; these approaches are discussed in §2.6. Verification is the topic of §2.7. Finally, in §2.8 we summarise and mention briefly some other approaches to modelling, specification and verification which have appeared in the literature.

2.2 Models of Time

Notation. In this section, and throughout the dissertation, the following notation is used to denote sets of numbers: $\mathbb{R}$ – the set of non-negative real numbers;

$\mathbb{Q}$ – the set of rational numbers; $\mathbb{Z}$ – the set of integers; and $\mathbb{N}$ – the set of natural numbers.

The model of time used in this work is the non-negative reals, which we denote by $\mathbb{R}$ and use with the usual operations of equality ($=$), ordering ($\leq$), addition ($+$), and multiplication ($\cdot$). As usual, we write $t < t'$ if $t \leq t'$ and $t \neq t'$. It is sometimes convenient to augment this domain with a value, ∞, which is defined to be strictly greater than any other time value. We write $\mathbb{R}_\infty$ for $\mathbb{R} \cup \{\infty\}$ and assume that the arithmetic operators and relations are extended to $\mathbb{R}_\infty$ in the usual way: for every $t \in \mathbb{R}$, $t < \infty$, and for every $t \in \mathbb{R}_\infty$, $t + \infty = \infty + t = \infty$. We also make use of an operator for subtraction, $\dot{-} : \mathbb{R}_\infty \times \mathbb{R} \to \mathbb{R}_\infty$, which satisfies,

$$t_1 \dot{-} t_2 = \begin{cases} 0 & \text{if } t_1 < t_2 \\ t & \text{if } t_2 \leq t_1 \wedge t_1 = t_2 + t. \end{cases}$$

This model of time is one of a number which have been proposed for use in the analysis of real-time systems [AH91, Jos91, Koy91, Nic92]. We briefly draw attention to some salient features and their relationship to the model of computation which will be used.

An important choice is the one between a *dense* or a *discrete* time domain. In a dense domain, such as $\mathbb{R}$ or $\mathbb{Q}$, any two distinct time points are separated by a set of intervening points which are also elements of the domain. In a discrete domain, such as $\mathbb{Z}$, each time point has a unique successor. Formally, $\mathbb{R}$ is a dense domain since it satisfies

$$(\exists\, t, t' \in \mathbb{R} \,.\, t < t') \wedge (\forall\, t, t' \in \mathbb{R} \mid t < t' \,.\, \exists\, t'' \in \mathbb{R} \,.\, t < t'' < t')$$

whereas $\mathbb{N}$ is a discrete domain since it satisfies

$$\forall\, t, t' \in \mathbb{N} \,.\, t < t' \Rightarrow (\exists\, t'' \in \mathbb{N} \,.\, t < t'' \wedge \forall\, t''' \in \mathbb{N} \,.\, t < t''' \Rightarrow t'' \leq t''').$$

Alur [Alu91] has argued convincingly that dense time is more appropriate in the modelling of *asynchronous* systems, where an arbitrarily small amount of time may separate event occurrences. If a discrete domain is chosen, then continuous physical time must be approximated by fixing a time *granularity* a priori, and no matter how fine the granularity chosen, for some systems the discrete model is not accurate enough to ensure that all possible erroneous behaviours will be detected [ACD93]. This problem has been noted also by Asarin et al. [AMP98] who exhibit a class of cyclic circuits as an example. Moreover, even when it is possible to choose a sufficiently fine granularity, it may be so fine that the size of the state space becomes too large for verification to be feasible. A dense domain is also more convenient when it comes to the composition of systems, since there is no need to worry about matching the time granularities of the components, as is the case for a discrete model. A possible advantage of the discrete model is that it facilitates the application of efficient verification techniques known from the analysis of untimed systems, in particular symbolic state space representation using binary decision diagrams

(BDDs) [Bry86, McM92]. It remains to be seen whether or not efficient symbolic representations will be discovered for dense time systems; the clock difference diagrams of [LWYP98] show some promise in this respect. Another interesting approach is to consider when it is possible to construct a discrete time model which is known to preserve dense time properties, since then we can have the expressiveness of the dense time model together with the efficient analysis of the discrete model [ABK+97, AMP98, BMPY97, HMP92].

An alternative to a *point-based* domain, such as $\mathbb{R}$, is a domain based on *intervals*, in which statements concerning the duration of events may be more conveniently expressed, see [Koy91] for further details. In its favour, we find that the domain $\mathbb{R}$ fits naturally with a simple computational model of time-stamped event sequences or trees. In this model, events are assumed to happen instantaneously, and system behaviour consists in a sequence of *two-phase steps*. In the first phase of a step, time passes by some finite or infinite amount. In the second phase, a finite, though arbitrarily large, number of instantaneous events occur in some well-defined order. A new step begins when the second phase terminates. This two-phase model has proven very effective in practice and is widely used; further arguments in its defence can be found in [NS91]. In this approach, a duration can be modelled by introducing instantaneous events representing its beginning and end.

It is convenient to assume that event sequences respect a *weakly monotonic* ordering, i.e., for a sequence $\langle (e_1, t_1), (e_2, t_2), \ldots \rangle$, where e_i represents an event and t_i its time-stamp, then t_i is required to be *less than or equal to* t_{i+1}, rather than *strictly less than*, as would be required by a *strongly monotonic* ordering. This allows concurrency to be modelled by the interleaving of events: for example, a computation in which the events a and b occur concurrently, can be modelled by the pair of sequences $\langle \ldots, (a, t_i), (b, t_{i+1}), \ldots \rangle$ and $\langle \ldots, (b, t_i), (a, t_{i+1}), \ldots \rangle$, where $t_i = t_{i+1}$ in each case.

One further point about the structure of time, which is also intimately related to the underlying computational model, concerns views of time as either a *linear* or a *branching* structure [EH86, Lam80, Pnu85]. In the linear model of time, it is assumed that at any moment there is only one possible next moment; system behaviour is represented as a set of possible execution sequences. In the branching model, time has a tree-like structure where it is assumed that each moment has at most one directly preceding moment, but perhaps many next moments, representing different possible futures; system behaviour is represented as a tree and an execution is a path through the tree. Each view supports the statement of system properties which cannot be expressed in the other. We regard the two views as complementary and make no commitment to either, but use whichever seems appropriate in the circumstances.

2.3 Transition Systems

2.3.1 Labelled Transition Systems

A method of modelling systems and their behaviour, which has been successfully applied in a wide variety of circumstances, is based on the idea that it is possible to identify a set of *states* which characterise certain aspects of the system which are of interest to the modeller. A system begins its operation in some *initial state*. During the operation of the system, its state may change. A change of state is called a *transition* and a system model consisting of states and transitions is a *state transition system* (usually abbreviated to *transition system*). It is often useful to associate a *label* with a transition. The label can be used for a variety of purposes: perhaps to identify an *action* which has caused the transition, or an *event* whose occurrence is indicated by the transition. A transition system in which labels are associated with transitions is called a *labelled transition system* (LTS). Within this basic framework, a system modeller has wide discretion in the choice of states, transitions and labels in the construction of a useful model. These ideas are presented formally below.

Definition 2.1 (Labelled Transition System) A *labelled transition system* $\mathcal{S} = (\Sigma, \sigma^{\mathcal{I}}, L, \longrightarrow)$ is a tuple where Σ is the set of states, $\sigma^{\mathcal{I}} \in \Sigma$ is the initial state, L is the set of labels and $\longrightarrow \subseteq \Sigma \times L \times \Sigma$ is the set of transitions. $\square$

Notation. We write $\sigma \xrightarrow{\lambda} \sigma'$ for $(\sigma, \lambda, \sigma') \in \longrightarrow$. If $\sigma \xrightarrow{\lambda} \sigma'$ for some label $\lambda \in L$ then σ' is said to be a λ-*successor* of σ and σ is a λ-*predecessor* of σ'. If σ' is a λ-successor (resp. -predecessor) of σ, then σ' is a *successor* (resp. *predecessor*) of σ. If σ has a λ-successor, we note this by $\sigma \xrightarrow{\lambda}$. If σ has no λ-successor, we write $\sigma \not\xrightarrow{\lambda}$. We use $\sigma_0 \longrightarrow^{\mathrm{n}} \sigma_n$ to denote $\sigma_0 \xrightarrow{\lambda_0} \sigma_1 \xrightarrow{\lambda_1} \cdots \xrightarrow{\lambda_{n-2}} \sigma_{n-1} \xrightarrow{\lambda_{n-1}} \sigma_n$, for $0 \leq i < n$ and $\lambda_i \in L$, and $\sigma_0 \longrightarrow^{*} \sigma_f$ if $\sigma_0 \longrightarrow^{\mathrm{n}} \sigma_f$ for some $n \in \mathbb{N}$.

Definition 2.2 (Finite, Finitely Branching, Deterministic) A transition system, $\mathcal{S} = (\Sigma, \sigma^{\mathcal{I}}, L, \longrightarrow)$, is *finite* if the set of states Σ and the transition relation $\longrightarrow$ are finite. $\mathcal{S}$ is *finitely branching* if for all $\sigma \in \Sigma$ and $\lambda \in L$, the set $\{(\lambda, \sigma') \mid \sigma \xrightarrow{\lambda} \sigma'\}$ is finite. $\mathcal{S}$ is *deterministic* if, for any state σ and label λ, if $\sigma \xrightarrow{\lambda} \sigma'$ and $\sigma \xrightarrow{\lambda} \sigma''$ then $\sigma' = \sigma''$. $\square$

Definition 2.3 (Isomorphism)
Let $\mathcal{S}_1 = (\Sigma_1, \sigma_1^{\mathcal{I}}, L, \longrightarrow_1)$ and $\mathcal{S}_2 = (\Sigma_2, \sigma_2^{\mathcal{I}}, L, \longrightarrow_2)$ be transition systems. $\mathcal{S}_1$ and $\mathcal{S}_2$ are said to be *isomorphic* iff there exists a bijection $f : \Sigma_1 \to \Sigma_2$ such that

1. $f(\sigma_1^{\mathcal{I}}) = \sigma_2^{\mathcal{I}}$, and

2. for every $\sigma, \sigma' \in \Sigma_1, \lambda \in L$, $\sigma \xrightarrow{\lambda}_1 \sigma'$ iff $f(\sigma) \xrightarrow{\lambda}_2 f(\sigma')$ $\square$

Definition 2.4 (Path) Let $\mathcal{S} = (\Sigma, \sigma^{\mathcal{I}}, L, \longrightarrow)$ be a transition system. Let $\sigma \in \Sigma$. A *path* in $\mathcal{S}$ *from* σ is a finite or infinite sequence, $\mathbf{p} = \sigma_0\lambda_0\sigma_1\lambda_1\sigma_2\lambda_2\cdots$, of alternating states and labels which satisfies

1. $\mathbf{p}$ starts with state $\sigma = \sigma_0$, known as the *source* of $\mathbf{p}$, and

2. for all $i = 0, 1, \ldots$, σ_{i+1} is a λ_i-successor of σ_i. □

A path of *length* n is a finite path $\mathbf{p} = \sigma_0\lambda_0\sigma_1\lambda_1 \cdots \lambda_{n-1}\sigma_n$. Let $\mathbf{p} = \sigma_0\lambda_0\sigma_1\lambda_1 \cdots$ be a finite or infinite path. For $i = 0, 1, 2, \ldots$, the *i-th state* of $\mathbf{p}$, denoted $\mathbf{p}(i)$, is defined to be σ_i and the *i-th label* of $\mathbf{p}$, denoted $\mathsf{label}_{\mathbf{p}}(i)$, is defined to be λ_i.

Definition 2.5 (Reachability) A state σ' is *reachable* from state σ iff there is a path in $\mathcal{S}$ from σ which contains σ'. The state σ is *reachable in* $\mathcal{S}$ iff σ is reachable from the initial state, $\sigma^{\mathcal{I}}$. □

2.3.2 Timed Transition Systems

A real-time system can be modelled as a labelled transition system. The actions of the system are represented by transitions whose labels are drawn from some set A of actions. Such transitions are known as *discrete transitions* and are assumed to be atomic and instantaneous. The passage of time is modelled by transitions whose labels are drawn from the set of non-negative real numbers $\mathbb{R}$; these transitions are called *time transitions*. The set of labels is thus $A \cup \mathbb{R}$. We assume $A \cap \mathbb{R} = \emptyset$. In order to serve as a model of a real-time system, we require that the transition system $\mathcal{S} = (\Sigma, \sigma^{\mathcal{I}}, L, \longrightarrow)$ satisfies the following properties:

Time determinism The evolution of the system is *deterministic* with respect to the passage of time [NS91, Nic92, Yi90], i.e., for a given state and a given time, there is at most one state which can be reached in a single step by taking the time transition. Formally,

$$\forall\, \sigma, \sigma', \sigma'' \in \Sigma;\ t \in \mathbb{R} \,.\, \sigma \xrightarrow{t} \sigma' \wedge \sigma \xrightarrow{t} \sigma'' \Rightarrow \sigma' = \sigma''$$

Time additivity The evolution of the system is *continuous* with respect to the passage of time [NS91, Nic92, Yi90]. If a time transition is possible from some state, then all smaller time transitions are also possible. Formally,

$$\forall\, \sigma, \sigma' \in \Sigma;\ t, t' \in \mathbb{R} \,.\, \sigma \xrightarrow{t+t'} \sigma' \Leftrightarrow \exists\, \sigma'' \in \Sigma \,.\, \sigma \xrightarrow{t} \sigma'' \wedge \sigma'' \xrightarrow{t'} \sigma'$$

Definition 2.6 (Timed Transition System) A *timed transition system* $\mathcal{S} = (\Sigma, \sigma^{\mathcal{I}}, L, \longrightarrow)$ is a labelled transition system whose set of labels L is $A \cup \mathbb{R}$ for some set A such that $A \cap \mathbb{R} = \emptyset$, and which satisfies the properties of time determinism and time additivity. □

Definition 2.7 (Execution, Run) An *execution* or *run* of a timed transition system $\mathcal{S}$, starting from a state σ, is an infinite path in $\mathcal{S}$ from σ. We denote the set of all executions from σ by $\Xi_{\mathcal{S}}(\sigma)$, and by $\Xi_{\mathcal{S}} = \bigcup_{\sigma \in \Sigma} \Xi_{\mathcal{S}}(\sigma)$ the set of executions of $\mathcal{S}$. □

We are primarily interested in those runs which can be regarded as a model of some physical system. In particular, we wish to ensure that basic physical laws concerning Time are respected:

1. a system cannot act with infinite speed, and
2. a system cannot block the progress of Time.

These ideas are captured for a timed transition system in the definition of a *time-divergent* run.

Definition 2.8 (Time-divergent run, Non-Zeno system)
Let $\mathcal{S} = (\Sigma, \sigma^{\mathcal{I}}, L, \longrightarrow)$ be a timed transition system, $\xi \in \Xi_{\mathcal{S}}$ an execution in $\mathcal{S}$ and $i, n \in \mathbb{N}$. The *i-th delay* in ξ, denoted $\delta_{\xi}(i)$, is defined to be $\mathsf{label}_{\xi}(i)$ if $\mathsf{label}_{\xi}(i) \in \mathbb{R}$, otherwise $\delta_{\xi}(i)$ is 0. The *time elapsed* in ξ from $\xi(0)$ to $\xi(n)$, denoted $\Delta_{\xi}(n)$, is defined

$$\Delta_{\xi}(n) = \sum_{i<n} \delta_{\xi}(i)$$

A run ξ is *time-divergent* (or simply *divergent*) iff $\lim_{i\to\infty} \Delta_{\xi}(i) = \infty$. The set of time-divergent runs from $\sigma \in \Sigma$ is denoted $\Xi^{\infty}_{\mathcal{S}}(\sigma)$ and the set $\bigcup_{\sigma\in\Sigma} \Xi^{\infty}_{\mathcal{S}}(\sigma)$ of all time-divergent runs in $\mathcal{S}$ is denoted $\Xi^{\infty}_{\mathcal{S}}$.

$\mathcal{S}$ is a *Non-Zeno* (*well-timed*) system iff every reachable state $\sigma \in \Sigma$ is the source of some time-divergent run. □

Remark 2.1 (Finite Variability, Time Progress) It follows directly from Definition 2.8 that there are a finite number of transitions represented in any bounded time interval of a divergent run, ξ. It is also apparent that for any $t \in \mathbb{R}$, there is a number $n \in \mathbb{N}$ such that $\Delta_{\xi}(n) > t$, i.e., time progresses beyond any bound.

2.3.3 Composition of transition systems

A complex system can be modelled by identifying and modelling smaller components of the whole system and then by stating precisely what is the behaviour of the system which is obtained by combining components.

A standard form of combination for transition systems is a *product* which models the parallel execution of two or more transition systems as a single system. We now define a commonly used product of transition systems. Let $\mathcal{S}_1 = (\Sigma_1, \sigma_1^{\mathcal{I}}, L_1, \longrightarrow_1)$ and $\mathcal{S}_2 = (\Sigma_2, \sigma_2^{\mathcal{I}}, L_2, \longrightarrow_2)$ be two transition systems which we assume to represent system components. In the product of $\mathcal{S}_1$ and $\mathcal{S}_2$, a state is a pair (σ_1, σ_2) where $\sigma_1 \in \Sigma_1$ and $\sigma_2 \in \Sigma_2$. The transitions of the product take their labels from the set $L_1 \cup L_2$. If λ is a label which occurs *both* in L_1 *and* in L_2, then we require each of $\mathcal{S}_1$ and $\mathcal{S}_2$ to perform a λ-labelled transition together in order for the product to perform a λ-labelled transition. If the label λ occurs in the set of labels of only one component, then that component can perform a λ-labelled transition *independently* in the product. The systems are said to *synchronise* on their shared labels, otherwise they act independently.

Definition 2.9 (Product of transition systems)
Let $\mathcal{S}_1 = (\Sigma_1, \sigma_1^{\mathcal{I}}, L_1, \longrightarrow_1)$ and $\mathcal{S}_2 = (\Sigma_2, \sigma_2^{\mathcal{I}}, L_2, \longrightarrow_2)$ be two transition systems. The *transition system product* of $\mathcal{S}_1$ and $\mathcal{S}_2$, which is written $\mathcal{S}_1 \mid \mathcal{S}_2$, is the transition system $(\Sigma_1 \times \Sigma_2, (\sigma_1^{\mathcal{I}}, \sigma_2^{\mathcal{I}}), L_1 \cup L_2, \longrightarrow)$ where $(\sigma_1, \sigma_2) \xrightarrow{\lambda} (\sigma_1', \sigma_2')$ iff

1. $\lambda \in L_1 \cap L_2$ and $\sigma_1 \xrightarrow{\lambda}_1 \sigma_1'$ and $\sigma_2 \xrightarrow{\lambda}_2 \sigma_2'$, or
2. $\lambda \in L_1 \setminus L_2$ and $\sigma_1 \xrightarrow{\lambda}_1 \sigma_1'$ and $\sigma_2' = \sigma_2$, or
3. $\lambda \in L_2 \setminus L_1$ and $\sigma_2 \xrightarrow{\lambda}_2 \sigma_2'$ and $\sigma_1' = \sigma_1$. □

2.4 Process Algebra

2.4.1 Basic concepts

The understanding of distributed systems has been advanced considerably by the study of *process algebra*. In this approach, a system is regarded as a process, which is constructed from smaller processes using a set of process constructors (operators). Some processes are regarded as primitive – not subject to further investigation – and larger processes are constructed from them using the process operators, resulting in an algebraic structure. Processes are investigated by considering equivalences between them, which leads to an equational style of reasoning. There are several different approaches to the algebraic treatment of processes. They can be characterised by:

- the choice of basic processes and process operators,
- the methods and models used to give a meaning to processes, and
- the notion of equivalence between processes.

The well known process algebras CCS [Mil89], CSP [Hoa85] and ACP [BW90] exemplify the main variations within each of these categories; these references should be consulted for a thorough introduction to the field. Here we mention some aspects which may be helpful in understanding the rest of the dissertation.

In process algebra, system events are modelled as atomic actions. In the family of ACP algebras, atomic actions are basic processes and act as the constants of the algebra. There is a sequential composition operator which models the execution of one process followed by the execution of another process. CCS adopts a different approach in which an atomic action a is not regarded as a basic process in its own right, but can be composed with some process P using an action prefix operator, to yield a new process $a.P$, which is capable of first performing the action a and then behaving as process P. In this approach, the **nil** process, which cannot perform any action, serves as a basic process. Given the possibility for modelling very simple systems such as these, more complex systems can be constructed using a variety of other operators including: choice, disabling, parallel composition and abstraction. Other features of system behaviour can also be modelled within the process algebraic framework, e.g., process priority, memory state and shared resources [BV95, LBGG94]. The formal description technique LOTOS [ISO88b] offers both a variety of useful process operators and a data language for modelling the data values which are stored and communicated by a system. It has been used extensively for modelling and analysing systems of practical interest.

Currently, the predominant method for giving a meaning to the terms of a process algebra is *structural operational semantics* (SOS) [Plo81]. SOS generates a labelled transition system, whose states are the terms of the process algebra, and whose transitions are obtained inductively from a set of transition rules of the form $\frac{\text{premises}}{\text{conclusions}}$. An example of a typical transition rule is

$$\frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'}$$

from which we can conclude the existence of an a-labelled transition from any term of the form $P + Q$ to a term of the form P', if we can demonstrate the existence of an a-labelled transition from P to P'. In general, validity of the premises of a transition rule, under a certain substitution, implies the validity of the conclusion of this rule under the same substitution [AFV99]. This operational style of semantic definition gives a meaning to a process description in terms of its effect upon the behaviour of some abstract machine. Other semantic approaches are the denotational method of CSP [BHR84] and the axiomatic method of ACP [BK84].

A variety of process equivalences are studied in the literature [vG90, vG93]. They range from a weak equivalence, in which processes are equated iff they can perform the same set of transition sequences, to a strong equivalence in which they are equated iff their derivation trees are isomorphic. The former equivalence may equate processes P and Q even though there are environments in which P deadlocks while Q does not. The latter equivalence may distinguish processes even if they can perform the same actions in all environments. Useful equivalences are found somewhere between these extremes. The variety of useful equivalences is greater in settings which distinguish between a set of actions which are *observable* and a set of actions which are *hidden* or *silent* [vG93]. The process equivalence of most relevance to our work is based on the idea of *strong bisimulation* [Mil89] and equates processes P and Q iff for every action a, every a-successor of P is equivalent to some a-successor of Q, and vice versa (cf. §3.6.3). This is generally regarded as the strongest of the useful equivalences. To be really useful, an equivalence should also be a *congruence*, i.e., equivalent processes should behave the same in all contexts, e.g., assume *op* is an arbitrary process operator and P and Q are equivalent processes, then $op(P_1, \ldots, P_{i-1}, P, P_{i+1} \ldots P_n)$ and $op(P_1, \ldots, P_{i-1}, Q, P_{i+1} \ldots P_n)$ should also be equivalent processes.

2.4.2 Timed Extensions

In the process algebras considered so far, there is not the possibility to model and reason about the quantitative aspects of the passage of time. This deficiency has been addressed by many researchers and, consequently, there are now many timed process algebras which can be used in the analysis of real-time systems. Vereijken [Ver97a] gives a very comprehensive review which covers almost 40 different timed process algebras. Nicollin and Sifakis [NS91] present a helpful unifying framework. Corradini et al. [CDI99] give a detailed study of the relationship between four CCS-like variants. Here we aim to give just a flavour of the main themes.

In general, timed process algebras introduce constants ranging over some time domain, either discrete or dense, and a number of time constraining operators, into the framework of an untimed algebra. A typical time constraining operator is one which delays a process, e.g., let t be a constant of the time domain, $t > 0$, then the process $(t).P$ is one which behaves just like P after exactly t time units. Such an operator is used in Temporal CCS [MT90], Timed CCS [Yi90], Real-Time CSP [Dav93] and Urgent LOTOS [BL91]. ACP$_\rho$ [BB91] adopts a different approach in which actions are time-stamped. Time stamps can be absolute or relative. In the absolute case, $a(t)$ performs the action a after t time units following the start of the process; in the relative case, $a[t]$ performs a after t time units following the execution of the previous action. The time-stamp operator has the effect of allowing the modelling both of delays and also of urgent actions; a delayed action becomes urgent when the time delay expires. Urgency can also be modelled by the introduction of immediate actions, which do not admit the possibility of time passing until either they are executed or disabled. This approach is adopted in ATP [NS94]. Other time constraining operators which have appeared in several algebras, and which are of practical interest for modelling real-time systems, are the timeout and watchdog operators. Real-time CSP offers both operators. Each takes two process arguments P and Q and a time parameter t. The timeout $P \rhd \{t\}\ Q$ behaves as P if an initial action of P is performed within time t, otherwise it behaves as Q, after time t. The watchdog $P \swarrow\{t\}\ Q$ behaves as P until time t. At time t, P is aborted and Q is started. Similar operators are found in other algebras, e.g. ATP.

Schneider [Sch95] discusses the operational, denotational and axiomatic styles of semantic definition in timed process algebras, and surveys the associated approaches to process equivalence. The decidability of timed bisimulation is shown in [Cěr92].

We return to some of the ideas mentioned in this section in Chapter 3, where their influence on the design of the language which is introduced there will be evident.

2.5 Timed Automata

2.5.1 Introduction

One of the most successful research areas of the last few years, in the modelling and analysis of real-time systems, features the use of *timed automata*, which were introduced in the seminal paper of Alur and Dill [AD90]. Early work concentrated on the theoretical aspects of the decidability and complexity of the model-checking and satisfiability problems for timed temporal logics such as TCTL [ACD90, AD94, AH91, Alu91]. Later, attention turned to the development of practical algorithms [HNSY94, YPD94]. More recently, the application of timed automata to the modelling of industrial problems [HSLL97, LPY98, TY98], and the development of software tools to support their analysis [BLL$^+$98, BDM$^+$98], have been receiving considerable attention.

Informally, a timed automaton is a finite state automaton in which the system states are augmented by a finite number of real-valued variables called *clocks*. All clocks are synchronised and are assumed to keep perfect time. Transitions between states can be constrained to occur when the values of the clocks satisfy some specified property. On the occurrence of a transition, one or more clocks can be reset to zero. In this way, it is possible to model the "real time" of occurrence of events and the time elapsed between events. Timed automata are presented formally below.

2.5.2 Clocks

Let $\mathcal{H}$ be a finite set of real-valued variables called *clocks*. A *$\mathcal{H}$-valuation* (clock valuation) is a total function $\mathbf{v} : \mathcal{H} \rightarrow \mathbb{R}$ which assigns to each clock $h \in \mathcal{H}$ a non-negative real number $\mathbf{v}(h)$. The set of $\mathcal{H}$-valuations is denoted $\mathbb{R}^{\mathcal{H}}$. The $\mathcal{H}$-valuation which assigns 0 to every clock in $\mathcal{H}$ is denoted $\mathbf{0}$. Let $\mathbf{v} \in \mathbb{R}^{\mathcal{H}}$ and $\mathsf{H} \subseteq \mathcal{H}$. $\mathbf{v}[\mathsf{H} := 0]$ denotes the valuation $\mathbf{v}'$ such that for all $h \in \mathcal{H}$, $\mathbf{v}'(h)$ is 0 if $h \in \mathsf{H}$ and is $\mathbf{v}(h)$ otherwise. This models the operation of resetting some clocks while leaving the values of the other clocks unchanged. The elapse of time is modelled by advancing the values of all clocks in a valuation by the same amount. Let $\mathbf{v} \in \mathbb{R}^{\mathcal{H}}$ and $t \in \mathbb{R}$. $\mathbf{v} + t$ denotes the valuation $\mathbf{v}'$ in which $\mathbf{v}'(h) = \mathbf{v}(h) + t$ for all clocks $h \in \mathcal{H}$. Occasionally, we will need the operation $t \cdot \mathbf{v}$ where for $t \in \mathbb{R}$ and $\mathbf{v} \in \mathbb{R}^{\mathcal{H}}$, $t \cdot \mathbf{v}$ is the valuation $\mathbf{v}'$ such that $\mathbf{v}'(h) = t \cdot \mathbf{v}(h)$, for all $h \in \mathcal{H}$.

2.5.3 Clock Constraints

Let $\mathcal{H}$ denote a set of clocks ranged over by h, h'. An *atomic constraint* on $\mathcal{H}$ is an expression of the form $h \bowtie c$ or $h - h' \bowtie c$, where $\bowtie \in \{<, \leq, \geq, >\}$ and $c \in \mathbb{N}$. The set of *clock constraints* on $\mathcal{H}$, denoted $\Psi_{\mathcal{H}}$, is generated by the grammar:

$$\psi ::= \chi \mid \psi \wedge \psi \mid \neg\, \psi$$

where χ is an atomic constraint. The set of *clock zones* on $\mathcal{H}$, denoted $\mathcal{Z}_{\mathcal{H}}$, with $\mathcal{Z}_{\mathcal{H}} \subset \Psi_{\mathcal{H}}$, is the set of conjunctions of atomic constraints. Let ζ, ζ' range over $\mathcal{Z}_{\mathcal{H}}$.

The restricted grammar of clock constraints is necessary in order to ensure that some important verification questions, such as model-checking, remain decidable. It is possible to extend the range of c to the non-negative rational numbers $\mathbb{Q}^+$, but the restriction to $\mathbb{N}$ simplifies the presentation at no cost to expressive power [AD94].

A clock valuation $\mathbf{v} \in \mathbb{R}^{\mathcal{H}}$ is said to *satisfy* a clock constraint $\psi \in \Psi_{\mathcal{H}}$, denoted $\mathbf{v} \models \psi$, if

$$\begin{array}{ll}
\mathbf{v} \models h \bowtie c & \text{iff } \mathbf{v}(h) \bowtie c \\
\mathbf{v} \models h - h' \bowtie c & \text{iff } \mathbf{v}(h) - \mathbf{v}(h') \bowtie c \\
\mathbf{v} \models \psi \wedge \psi' & \text{iff } \mathbf{v} \models \psi \text{ and } \mathbf{v} \models \psi' \\
\mathbf{v} \models \neg\, \psi & \text{iff } \mathbf{v} \not\models \psi
\end{array}$$

The set of all clock valuations satisfying a clock constraint $\psi \in \Psi_{\mathcal{H}}$ is denoted $[\![\psi]\!]$, i.e., $[\![\psi]\!] = \{\mathbf{v} \in \mathbb{R}^{\mathcal{H}} \mid \mathbf{v} \models \psi\}$.

We use tt to denote a clock constraint such as $h \geq 0$ which is satisfied by any clock valuation, and ff to denote a clock constraint such as $h < 0$ which is not satisfied by any clock valuation, i.e., $[\![\mathsf{tt}]\!] = \mathbb{R}^{\mathcal{H}}$ and $[\![\mathsf{ff}]\!] = \emptyset$. It is also useful to have a notation for the clock constraint which requires that all clocks have the value 0, $\mathsf{zero}_{\mathcal{H}}$ denotes such a constraint, i.e., $\mathsf{zero}_{\mathcal{H}} \mathrel{\widehat{=}} \bigwedge_{h \in \mathcal{H}} h = 0$, (we will just write zero when $\mathcal{H}$ is clear from the context).

2.5.4 Syntax and informal semantics

We can now give a formal definition of the syntax of timed automata. We also provide some simple examples and an informal explanation of semantics.

Definition 2.10 (Timed Automaton) A *timed automaton* (TA) is a tuple $\mathcal{A} = (Q, q^{\mathcal{I}}, A, \mathcal{H}, E, I)$ where:

- Q is a finite set of *control locations.*
- $q^{\mathcal{I}} \in Q$ is the initial control location.
- A is a finite set of action labels.
- $\mathcal{H}$ is a finite set of clocks.
- $E \subseteq Q \times \mathcal{Z}_{\mathcal{H}} \times A \times 2^{\mathcal{H}} \times Q$ is a finite set of edges.

  Each edge $e \in E$ is of the form $(q, \zeta, a, \mathsf{H}, q')$ where $q, q' \in Q$ are control locations, denoted $\mathsf{src}(e), \mathsf{tgt}(e)$, respectively; $\zeta \in \mathcal{Z}_{\mathcal{H}}$ is a clock zone, called the *guard* of e and denoted $\mathsf{guard}(e)$; $a \in A$ is an action label, denoted $\mathsf{label}(e)$ and $\mathsf{H} \subseteq \mathcal{H}$ is a set of clocks to be reset, denoted $\mathsf{reset}(e)$.
- $I : Q \to \mathcal{Z}_{\mathcal{H}}$ is a function which associates a *time progress* condition (or *invariant*) with each control location. Control can remain at a location while time passes so long as the invariant associated with the location remains true.

We use $c_{max}(\mathcal{A}, h)$ to denote the greatest constant to which the clock variable h is compared in any guard or invariant condition of $\mathcal{A}$, and $c_{max}(\mathcal{A})$ to denote $\mathsf{max}\{c_{max}(\mathcal{A}, h) \mid h \in \mathcal{H}\}$. □

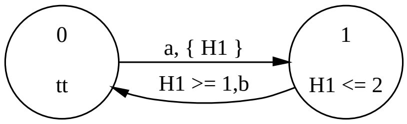


Fig. 2.1: A simple timed automaton

Example 2.1 We can explain some of these details informally by reference to Figure 2.1 which shows a simple example of a TA. The set of control locations is $\{0, 1\}$. Location 0 is assumed to be the initial location. The set of action labels is $\{a, b\}$. The set of clocks is $\{H1\}$. The invariant associated with location 0 is $\mathtt{tt}$; this means that the system can spend an arbitrary amount of time in location 0. In the absence of an explicit clock constraint, the edge from 0 to 1 is assumed to have the clock constraint $\mathtt{tt}$, and so an a-transition from 0 to 1 is possible at any time. If an a-transition occurs, clock $H1$ is reset to 0. While in location 1, the value of clock $H1$ shows the amount of time for which control has been at this location. Control can remain here for no more than 2 time units, as shown by the invariant $H1 \leq 2$, i.e., the invariant serves as a way of enforcing progress: some transition via an outgoing edge must be taken before the location invariant becomes false. The constraint $H1 \geq 1$ on the edge from 1 to 0 ensures that a b-transition cannot occur until control has resided at location 1 for at least 1 time unit, when a b-transition becomes possible, taking control back to location 0. It is assumed that no clocks are reset by a b-transition (the missing reset set on the edge from 1 to 0 is taken to be $\emptyset$). The timing requirement expressed by this automaton is that every a action is inevitably followed by a b action after a delay of 1 to 2 time units. $\square$

2.5.5 Formal Semantics

The semantics of the timed automaton, $\mathcal{A}$, is defined by assigning a timed transition system to it. A *state* in the transition system is a pair $(q, \mathbf{v})$ where q is a location of $\mathcal{A}$ and $\mathbf{v}$ is a clock valuation satisfying the invariant of q. The initial state consists of the initial location and the clock valuation in which all clocks are set to 0. The transition relation $\longrightarrow$ comprises both *discrete* transitions and *time* transitions. In a discrete transition, the location of control may change by following an outgoing edge. In a time transition, the location of control remains the same while time passes; the location invariant must be satisfied throughout the passage of time. Formally, the semantics is defined as follows:

Definition 2.11 (Timed Automaton Semantics) The semantics of the timed automaton $\mathcal{A} = (Q, q^{\mathcal{I}}, A, \mathcal{H}, E, I)$ is given by the timed transition system $\mathcal{T}[\![\mathcal{A}]\!] = (\Sigma, \sigma^{\mathcal{I}}, L, \longrightarrow)$ where

- $\Sigma = \{(q, \mathbf{v}) \mid q \in Q \land \mathbf{v} \in \mathbb{R}^{\mathcal{H}} \land \mathbf{v} \models I(q)\}$.

- $\sigma^{\mathcal{I}} = (q^{\mathcal{I}}, \mathbf{0})$ is the initial state.
- $L = A \cup \mathbb{R}$ is the set of labels.
- $\longrightarrow \subseteq \Sigma \times L \times \Sigma$ is the transition relation defined by:
  - *Discrete transitions*

$$\textbf{TA.1}\ \frac{(q, \zeta, a, \mathsf{H}, q') \in E \wedge \mathbf{v} \models \zeta \wedge \mathbf{v}[\mathsf{H} := 0] \models I(q')}{(q, \mathbf{v}) \xrightarrow{a} (q', \mathbf{v}[\mathsf{H} := 0])}$$

    We say that $(q', \mathbf{v}[\mathsf{H} := 0])$ is a *discrete successor* of $(q, \mathbf{v})$.

  - *Time transitions*

$$\textbf{TA.2}\ \frac{t \in \mathbb{R} \wedge \forall\, t' \in \mathbb{R} \mid t' \leq t\ .\ \mathbf{v} + t' \models I(q)}{(q, \mathbf{v}) \xrightarrow{t} (q, \mathbf{v} + t)}$$

    We say that $(q, \mathbf{v} + t)$ is a *time successor* of $(q, \mathbf{v})$. □

Notation. If $\sigma = (q, \mathbf{v})$, then $\sigma + t$ denotes the state $(q, \mathbf{v} + t)$ and $\sigma[\mathsf{H} := 0]$ denotes $(q, \mathbf{v}[\mathsf{H} := 0])$.

Example 2.2 Referring again to Figure 2.1, we can see that the state space $\Sigma \subseteq \{0, 1\} \times (\{H1\} \rightarrow \mathbb{R})$. The initial state is $(0, \{H1 \mapsto 0\})$. The label set $L = \{a, b\} \cup \mathbb{R}$ and some possible transitions are:

$$(0, \{H1 \mapsto 0\}) \xrightarrow{1.7} (0, \{H1 \mapsto 1.7\}) \xrightarrow{a} (1, \{H1 \mapsto 0\}) \xrightarrow{0.2} (1, \{H1 \mapsto 0.2\}) \xrightarrow{1.3}$$
$$(1, \{H1 \mapsto 1.5\}) \xrightarrow{b} (0, \{H1 \mapsto 1.5\}) \xrightarrow{50} (0, \{H1 \mapsto 51.5\}) \xrightarrow{a} (1, \{H1 \mapsto 0\}) \ldots$$

□

We define a notion of *deterministic* timed automata by analogy with the classical notion of determinism for finite state automata, viz., the state reached by following an edge with a given label is uniquely determined by the current state. However, in the case of timed automata, it is not necessary to prohibit the use of the same label on distinct outgoing edges of every location, but, instead, it is required only that for any pair of such edges, the associated clock constraints are mutually exclusive, so that at any time at most one of them is enabled.

Definition 2.12 (Deterministic Timed Automaton) A timed automaton is said to *deterministic* iff for all $q \in Q$, for all $a \in A$ and for every pair of distinct edges of the form $(q, \zeta_1, a, \mathsf{H}_1, q')$ and $(q, \zeta_2, a, \mathsf{H}_2, q'')$, there is no clock valuation $\mathbf{v}$ which satisfies *both* of the following conditions:

1. $\mathbf{v} \models \zeta_1$ and $\mathbf{v}[\mathsf{H}_1 := 0] \models I(q')$,
2. $\mathbf{v} \models \zeta_2$ and $\mathbf{v}[\mathsf{H}_2 := 0] \models I(q'')$. □

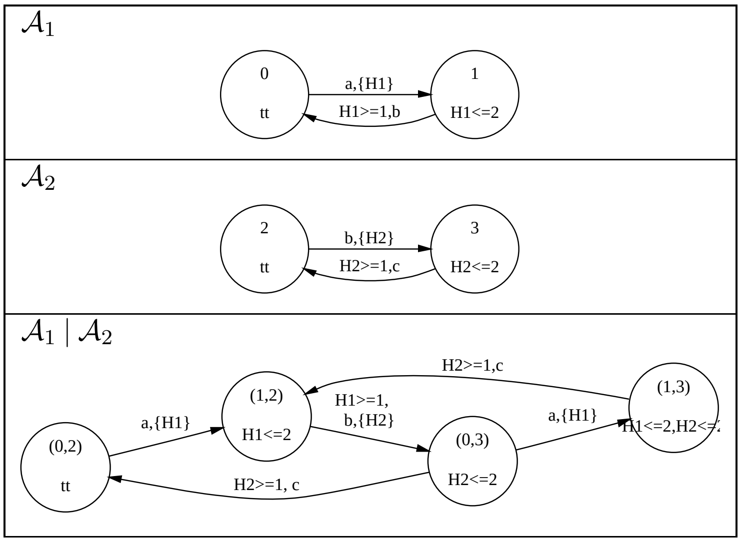


Fig. 2.2: Product construction for timed automata

2.5.6 Composition of timed automata

By defining a product for timed automata, we can model a complex system using several smaller, interacting component automata.

Let $\mathcal{A}_1 = (Q_1, q_1^{\mathcal{I}}, A_1, \mathcal{H}_1, E_1, I_1)$ and $\mathcal{A}_2 = (Q_2, q_2^{\mathcal{I}}, A_2, \mathcal{H}_2, E_2, I_2)$ be two timed automata. Assume that the clock sets $\mathcal{H}_1$ and $\mathcal{H}_2$ are disjoint. Then the product, denoted $\mathcal{A}_1 \mid \mathcal{A}_2$, is the timed automaton $(Q_1 \times Q_2, (q_1^{\mathcal{I}}, q_2^{\mathcal{I}}), A_1 \cup A_2, \mathcal{H}_1 \cup \mathcal{H}_2, E, I)$, where $I(q_1, q_2)$ is defined to be $I_1(q_1) \wedge I_2(q_2)$ and the edges E are given by:

1. For $a \in A_1 \cap A_2$, for every $(q_1, \zeta_1, a, \mathsf{H}_1, q_1') \in E_1$ and $(q_2, \zeta_2, a, \mathsf{H}_2, q_2') \in E_2$, E contains $((q1, q2), \zeta_1 \wedge \zeta_2, a, \mathsf{H}_1 \cup \mathsf{H}_2, (q_1', q_2'))$

2. For $a \in A_1 \setminus A_2$, for every $(q_1, \zeta, a, \mathsf{H}, q_1') \in E_1$ and every $q_2 \in Q_2$, E contains $((q_1, q_2), \zeta, a, \mathsf{H}, (q_1', q_2))$

3. For $a \in A_2 \setminus A_1$, for every $(q_2, \zeta, a, \mathsf{H}, q_2') \in E_2$ and every $q_1 \in Q_1$, E contains $((q_1, q_2), \zeta, a, \mathsf{H}, (q_1, q_2'))$

From this we can see that the locations of the product are just pairs of component locations and the invariant of a compound location is the conjunction of the invariants of its component locations. The edges are obtained by synchronising edges with identical labels.

For timed automata $\mathcal{A}_1$ and $\mathcal{A}_2$, it can be shown that the product of the models of $\mathcal{A}_1$ and $\mathcal{A}_2$ is the same as the model of the product of $\mathcal{A}_1$ and $\mathcal{A}_2$; i.e., $\mathcal{T}[\![\mathcal{A}_1]\!] \mid \mathcal{T}[\![\mathcal{A}_2]\!]$ is isomorphic to $\mathcal{T}[\![\mathcal{A}_1 \mid \mathcal{A}_2]\!]$ [AD94]. Figure 2.2 shows a simple example of a product construction of timed automata.

Example 2.3 (Train Gate Controller) The level crossing controller is a ubiquitous introductory example. We consider a simple system consisting of three

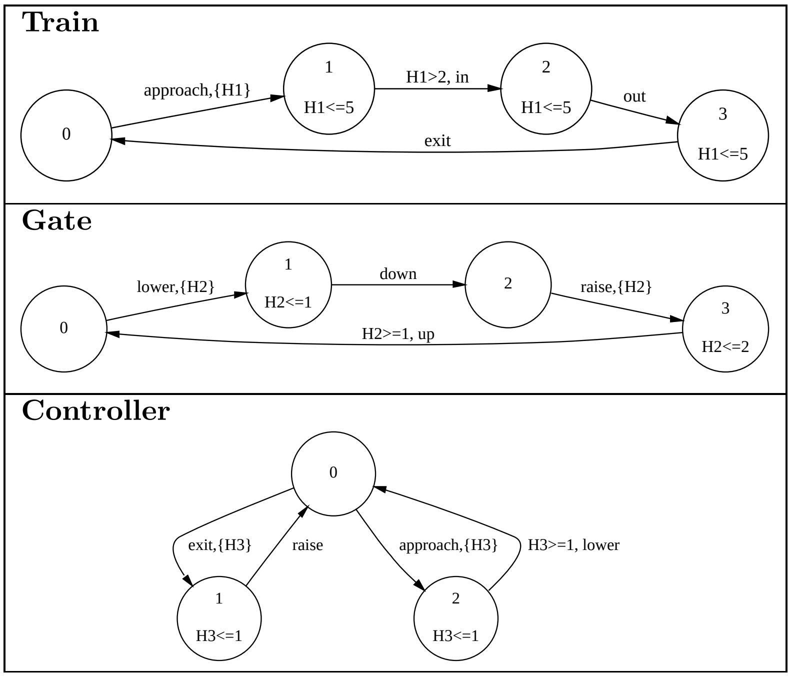


Fig. 2.3: Level Crossing Control System

components: a train, a gate and a controller. Each of these can be modelled as a TA (see Figure 2.3). Timing constraints are expressed using 3 clocks: $H1$ for the train, $H2$ for the gate and $H3$ for the controller. The train advises the controller of its approach more than 2 minutes before it enters the crossing. The approach of the train is indicated by the action `approach`, and entry into the crossing by the action `in`. Notice that the guard on the edge labelled `in` is `H1 > 2`. The maximum delay between the actions `approach` and `exit` is 5 minutes. The gate is open in location `Gate.0` and closed in location `Gate.2`. The actions `raise` and `lower` are used to indicate requests for service from the gate by the controller. The actions `up` (resp. `down`) indicate that the gate has been completely raised (resp. lowered). The controller idles in location `Controller.0`. Whenever it detects that the train is approaching, it requests that the gate should be lowered. Similarly, whenever it detects that the train has left the crossing, it requests that the gate should be raised. The complete system is expressed as the composition of the three components `Train | Gate | Controller`. The safety requirement for the system is straightforward: whenever the train is in the crossing, the gate should be closed. □

2.6 Property Specification

The main point of constructing a formal system model is to check it for the presence of desirable properties and the absence of undesirable properties. A first step in this direction involves formally stating the properties of interest. A classification of properties which has proved of enduring usefulness is the distinction between *safety* and *liveness* properties, introduced by Lamport [Lam77, Lam80]. Informally, a safety property specifies that 'nothing bad ever happens', while a liveness property specifies that 'something good eventually happens'. There is a variety of approaches to expressing both safety and liveness properties of timed transition system models. We consider some of them in this section. The remainder of the section is structured as follows. In §2.6.1, we consider the expression of properties of individual states using *state formulas*. This allows us to state simple safety invariants which can be checked by exploring all reachable states and testing them for satisfaction of the invariant property. More complicated properties, involving system executions, can be expressed using *specification automata* or *temporal logic*. These approaches are considered in §2.6.2 and §2.6.3, respectively. The relationship between automata and temporal logic is considered in §2.6.4.

2.6.1 State Properties

For a state transition model, $\mathcal{S}$, the simplest properties to assert and check are those concerned only with individual states, i.e. given some state σ determine whether or not a property p holds at σ. What structure we attribute to a state will depend on the circumstances. At the least, we assume that a state is associated with a unique identifier; sometimes, in addition, we assume that a state gives a valuation for a set of typed variables. Let *Var* be such a set and let x range over *Var*. The value of x at state σ is denoted $\sigma.x$. We assume that a *state formula* p is a boolean expression constructed in the usual way from variables, function symbols, predicate symbols and boolean connectives, and that there is a valuation function $[\![p]\!]_\sigma$, which gives the value of p at σ. We write $\sigma \models p$ iff $[\![p]\!]_\sigma = \mathbf{true}$. The reader should refer to [MP92] for further explanation of state formulas, if required.

Let $\mathcal{S} = \mathcal{T}[\![\mathcal{A}]\!]$ be a transition system, where the TA $\mathcal{A} = (Q, q^{\mathcal{I}}, A, \mathcal{H}, E, I)$ is either a simple TA, or a composition of TA $\mathcal{A}_1 \,|\, \cdots \,|\, \mathcal{A}_n$. Then $\mathsf{enable}(a)$ and $\mathcal{A}_i@q$ can be encoded as state formulas, where $a \in A$ is an action and $q \in Q$ is a location. Informally, $\mathsf{enable}(a)$ is true if it is possible to take an a-transition from the current state, and $\mathcal{A}_i@q$ is true if control in the TA $\mathcal{A}_i$ currently resides at location q.

Formally,

$$(q, \mathbf{v}) \models \mathsf{enable}(a) \text{ iff } \exists (q, \zeta, a, \mathsf{H}, q') \in E \,.\, \mathbf{v} \models \zeta \wedge \mathbf{v}[\mathsf{H} := 0] \models I(q')$$

and, for $1 \le i \le n$,

$$((q_1, q_2, \ldots, q_n), \mathbf{v}) \models \mathcal{A}_i@q_i.$$

Example 2.4 Let $\mathcal{S} = \mathcal{T}[\![\texttt{Train | Gate | Controller}]\!]$, where `Train`, `Gate` and `Controller` are as given in Figure 2.3. Let $q = (1, 1, 0)$, i.e. q is a compound location in which the components are: `Train` at location 1, `Gate` at location 1 and `Controller` at location 0. Let $\mathbf{v} = \{h_1 \mapsto 1.5, h_2 \mapsto 0.5, h_3 \mapsto 1.5\}$

be a clock valuation. Let $\sigma = (q, \mathbf{v})$. Then, we have $\sigma \models \mathsf{enable}(down)$ and $\sigma \models \mathtt{Gate}@1$, but $\sigma \not\models \mathsf{enable}(in)$ and $\sigma \not\models \mathtt{Controller}@1$. □

Example 2.5 Let σ be a state over integer variables x, y and boolean variable z, such that $\sigma = \{x \mapsto 5, y \mapsto 7, z \mapsto true\}$. Then, $\sigma \models z$, $\sigma \not\models x > y$, $\sigma \models x + y < 15$ and $\sigma \models z \Rightarrow (y - x = 2)$. □

Example 2.6 In the level crossing control system of Figure 2.3, the safety requirement can be stated as the absence of any reachable state σ satisfying $\sigma \models \mathtt{Train}@2 \wedge \neg\, \mathtt{Gate}@2$. □

2.6.2 Automata

We can go beyond checking simple state properties and check properties of executions by reasoning about the system in the context of a testing (observer) automaton. Given a TA $\mathcal{A}_M$ which models the behaviour of a system, a test TA $\mathcal{A}_S$ is constructed to capture a property specification, and the composition $\mathcal{A}_M \mid \mathcal{A}_S$ is checked to see if some error state is reachable. Using this technique it is possible, for example, to test a bounded response property, i.e., that the occurrence of a stimulus is followed by a response within a bounded period of time.

Example 2.7 Figure 2.4[1] shows a test automaton for the level crossing control system. The test automaton is used to check the bounded response property 'the gate is always raised strictly within 10 minutes of being lowered'. We consider the behaviour of the composition `Train | Gate | Controller | Test`. A `down` action in `Gate` synchronises with a `down` action in `Test`, causing a transition in `Test` to location 1, resetting the test clock `Ht`. The invariant `Ht <= 10` ensures that control can reside at `Test.1` for no more than 10 minutes. At any time before 10 minutes, an occurrence of any action other than `up` leaves control at `Test.1`; an `up` action returns control to `Test.0`. When 10 minutes have passed at `Test.1`, the only possible action for `Test` is `fail`, which takes control to the error location `Test.2`. The bounded response property for the system is satisfied iff it is not possible to reach a state satisfying `Test@2`. □

The approach to property checking via test automata is closely related to classical verification methods based on language containment [AD94, Kur94, Tho90]. We give a brief introduction to the use of such a method for timed systems.

Let $\mathcal{A}_M$ be a TA defining a system model and $\mathcal{A}_S$ a TA, extended with an *acceptance condition*, which defines a property specification. Let $\mathcal{L}_M =$

[1] Standard abbreviations are used in the figure to reduce its size: an edge labelled with a set of actions A represents a set of edges, one for each action in A; for any action $a \in A$, the notation $\setminus a$ stands for the set $A \setminus \{a\}$.

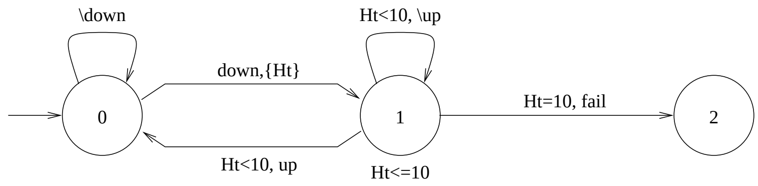


Fig. 2.4: `Test` automaton for bounded response

$\Xi^\infty(\sigma_M^{\mathcal{I}})$ be the set of non-Zeno executions of $\mathcal{A}_M$ and $\mathcal{L}_S = \{\xi \in \Xi^\infty(\sigma_S^{\mathcal{I}}) \mid \xi$ satisfies the acceptance condition of $\mathcal{A}_S\}$ the set of those non-Zeno executions of $\mathcal{A}_S$ which satisfy its acceptance condition. $\mathcal{L}_M$ is called the *language* of $\mathcal{A}_M$ and $\mathcal{L}_S$ the language of $\mathcal{A}_S$. The system model satisfies the specification iff $\mathcal{L}_M \subseteq \mathcal{L}_S$, i.e., if the language $\mathcal{L}_M \cap \overline{\mathcal{L}_S} = \emptyset$. Intuitively, $\mathcal{A}_S$ defines the set of all *allowed* executions and $\mathcal{A}_M$ defines the set of all *possible* executions of the system. The verification problem is to show that all possible executions are allowed or, equivalently, that no disallowed execution is possible. Attention is restricted to the non-Zeno executions since they are the only ones which can reasonably be judged to model the behaviour of a physical system.

Several acceptance conditions have been proposed in the literature [Tho90]. For timed systems, Büchi acceptance and Muller acceptance have received most attention [AD94]. Here we concentrate on Büchi acceptance.

Büchi acceptance is defined for a TA $\mathcal{A} = (Q, q^{\mathcal{I}}, A, \mathcal{H}, E, I)$ augmented with a set $F \subseteq Q$ of *accepting* locations. In any execution, one or more locations are visited infinitely often. Let $\mathsf{inf}(\xi)$ be the set of all infinitely occurring locations of the execution ξ. ξ is *accepted* iff $\mathsf{inf}(\xi) \cap F \neq \emptyset$, i.e. if some accepting state occurs infinitely often in it. A timed automaton extended with a Büchi acceptance condition is called a timed Büchi automaton (TBA).

In practice, the test for language containment $\mathcal{L}_M \subseteq \mathcal{L}_S$ is usually implemented by constructing the automaton $\mathcal{A}_M \mid \overline{\mathcal{A}_S}$ and checking for the absence of any acceptance cycle. A problem with this approach is the requirement to construct the complement $\overline{\mathcal{A}_S}$ of the specification automaton $\mathcal{A}_S$, since TBA are not closed under complementation. However, if $\mathcal{A}_S$ is deterministic then it can be complemented effectively. The restriction to deterministic TBA still allows the expression of a wide range of specifications. An even more pragmatic approach to the problem of complementation is to avoid it entirely by requiring the specifier to provide $\overline{\mathcal{A}_S}$ directly, rather than $\mathcal{A}_S$. This approach is adopted, for example, in [Tri98] where an efficient algorithm is given for testing TBA emptiness.

Example 2.8 Figure 2.5[2] shows a deterministic TBA which specifies the bounded response property for the level crossing control system. □

[2] Accepting locations are shown as a double circle, as usual.

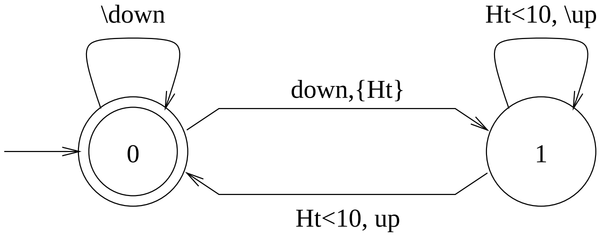


Fig. 2.5: TBA for bounded response

2.6.3 Temporal Logic

Temporal logic [Eme90] was developed originally in the field of philosophy, where it was used to describe and reason about how the truth values of assertions vary with time. For some assertion ϕ, typical temporal operators include *sometime* ϕ, which is true now if ϕ will become true at some time in the future, and *always* ϕ, which is true now if ϕ is true now and forever more. Pnueli [Pnu77] was the first to show how temporal logic could be used to reason about the behaviour of computer programs, particularly *reactive* programs such as operating systems and communication protocols. This early work often involved a difficult manual construction of the proof of some program property. Interest in the use of temporal logic for program specification increased when it was shown that the validity of a specification for a given program could be determined *automatically* by *model checking* [CE81, QS81], i.e., by checking the truth or falsehood of the specification when interpreted using the program as a model. The EMC model checker, developed at Carnegie Mellon, allowed small programs to be checked automatically in linear time for satisfaction of specifications written in the branching time logic CTL [CES86]. Activity in the area intensified with the introduction of *symbolic* methods [BCM$^+$92, McM92] which facilitate the storage of the large state spaces which arise in the checking of realistic programs. The extension of temporal logics with *explicit references to time quantities* was motivated by the desire to apply temporal logic to the specification and verification of real-time programs, where, for example, it is not enough to assert just "*sometime* ϕ", but rather "*sometime within the next 5 seconds* ϕ". Early quantitative temporal logics were based upon a discrete model of time [AH93, Eme91, EMSS90, HLP90, Ost86]. However, the decidability of the model-checking problem for a *dense time* model was demonstrated in [ACD90] which introduced Timed Computation Tree Logic (TCTL), a timed extension of CTL. The usefulness of this result was advanced by [HNSY94] which gave a practical method for implementing the model-checking of timed automata with respect to TCTL specifications; this method has been implemented in the verification tool KRONOS [BDM$^+$98]. An efficient, on-the-fly implementation of model-checking for TECTL$^*_\exists$, a logic strictly more expressive than TCTL, is proposed in [BTY97].

It is outside the scope of this dissertation to provide a detailed survey of temporal logics and model-checking, for which we refer the reader to the literature [AH91, CGP99, Eme90, Yov97]; however, we do provide an introduction to TCTL, since it is used in the rest of the dissertation for specifying real-time properties.

TCTL: Syntax and Semantics

Let $\mathcal{I}$ denote the set of all intervals of $\mathbb{R}$ of the form $[c,c'], [c,c'), (c,c'], (c,c')$, $[c,\infty]$ and (c,∞) where $c, c' \in \mathbb{N}$. The set of TCTL formulas is defined by the following syntax:

$$\phi ::= p \mid \neg\, \phi \mid \phi \vee \phi \mid \phi\ \exists\mathcal{U}_I\ \phi \mid \phi\ \forall\mathcal{U}_I\ \phi$$

where p is a state formula and $I \in \mathcal{I}$ is an interval.

Let $\mathcal{A}$ be a TA. TCTL formulas are interpreted with respect to the transition system $\mathcal{T}[\![\mathcal{A}]\!] = (\Sigma, \sigma^{\mathcal{I}}, L, \longrightarrow)$, and a satisfaction relation $\Vdash$ for state formulas p. The fact that a state $\sigma \in \Sigma$ satisfies a TCTL formula ϕ is denoted $\sigma \models_{(\mathcal{A},\Vdash)} \phi$ (the subscript is usually omitted to avoid clutter). The dense nature of the time model requires us to take some care in the definition of satisfaction for TCTL and it is helpful to introduce some further notation before giving a formal definition. For a state σ, the temporal modalities $\exists\mathcal{U}_I$ and $\forall\mathcal{U}_I$ are interpreted with respect to the non-Zeno executions starting from σ, i.e., $\Xi^{\infty}_{\mathcal{A}}(\sigma)$. Suppose $\xi \in \Xi^{\infty}_{\mathcal{A}}(\sigma)$ is such an execution, along which we see the partial sequence $\ldots \sigma_i \xrightarrow{t} \sigma_{i+1} \ldots$. In interpreting a formula such as $\phi_1\ \exists\mathcal{U}_I\ \phi_2$, we are required, by the dense nature of time, to consider the truth values of the sub-formulas ϕ_1 and ϕ_2, not only at σ_i and σ_{i+1}, but also at all states between them, as time passes for t time units. This motivates the introduction of the idea of a *position* along an execution, where for an execution $\xi \in \Xi_{\mathcal{A}}(\sigma)$ a *position* of ξ is a pair $(i,t) \in \mathbb{N} \times \mathbb{R}$ such that $t \le \delta_\xi(i)$. We denote by Π_ξ the set of all positions of ξ. Positions are ordered lexicographically so that $(i,t) \le (j,t')$ iff $i < j$, or $i = j$ and $t \le t'$. Given an execution ξ and a position (i,t) of ξ, we use $\xi(i,t)$ to denote the state $\xi(i) + t$, and $\Delta_\xi(i,t)$ to denote $\Delta_\xi(i) + t$. We can now define $\sigma \models \phi$ as follows:

$$\begin{array}{lll}
\sigma \models p & \text{iff} & \sigma \Vdash p \\
\sigma \models \neg\, \phi & \text{iff} & \sigma \not\models \phi \\
\sigma \models \phi_1 \vee \phi_2 & \text{iff} & \sigma \models \phi_1 \text{ or } \sigma \models \phi_2 \\
\sigma \models \phi_1\ \exists\mathcal{U}_I\ \phi_2 & \text{iff} & \exists\, \xi \in \Xi^{\infty}_{\mathcal{A}}(\sigma)\ .\ \exists\, \pi \in \Pi_\xi\ .\ \Delta_\xi(\pi) \in I \wedge \xi(\pi) \models \phi_2 \wedge \\
 & & \forall\, \pi' \le \pi\ .\ \xi(\pi') \models \phi_1 \vee \phi_2 \\
\sigma \models \phi_1\ \forall\mathcal{U}_I\ \phi_2 & \text{iff} & \forall\, \xi \in \Xi^{\infty}_{\mathcal{A}}(\sigma)\ .\ \exists\, \pi \in \Pi_\xi\ .\ \Delta_\xi(\pi) \in I \wedge \xi(\pi) \models \phi_2 \wedge \\
 & & \forall\, \pi' \le \pi\ .\ \xi(\pi') \models \phi_1 \vee \phi_2
\end{array}$$

A TA $\mathcal{A}$ is said to satisfy a TCTL formula ϕ, denoted $\mathcal{A} \models \phi$, if the initial state $\sigma^{\mathcal{I}}$ satisfies ϕ.

The only tricky parts in the definition of satisfaction concern the operators $\exists\mathcal{U}_I$ and $\forall\mathcal{U}_I$. The intention is that a state σ satisfies the formula $\phi_1\ \exists\mathcal{U}_I\ \phi_2$ if there is some position along a non-Zeno run starting from σ which satisfies ϕ_2, and the time elapsed in the run up to that position lies within the interval I, and finally that ϕ_1 is satisfied continuously throughout the run up to that position. In fact, the formal statement of the final condition is that $\phi_1 \vee \phi_2$ is satisfied continuously until ϕ_2 is satisfied. This modification is required to comply with the dense nature of the time domain, as explained in [HNSY94]. The interpretation for $\forall \mathcal{U}_I$ is similar, the only difference being that the conditions must be satisfied by *all* non-Zeno runs from σ.

A number of abbreviations are commonly used:

$$\begin{array}{rcl}
\exists\Diamond_I\,\phi & \mathrel{\widehat{=}} & \mathbf{true}\;\exists\mathcal{U}_I\;\phi \\
\forall\Diamond_I\,\phi & \mathrel{\widehat{=}} & \mathbf{true}\;\forall\mathcal{U}_I\;\phi \\
\exists\Box_I\,\phi & \mathrel{\widehat{=}} & \neg\;\forall\Diamond_I\,\neg\,\phi \\
\forall\Box_I\,\phi & \mathrel{\widehat{=}} & \neg\;\exists\Diamond_I\,\neg\,\phi
\end{array}$$

Other abbreviations are used to simplify the notation for intervals: for example, $\forall\Diamond_{\leq 5}\,\phi$ is equivalent to $\forall\Diamond_{[0,5]}\,\phi$ and $\exists\Box\,\phi$ is equivalent to $\exists\Box_{[0,\infty)}\,\phi$.

Property Specification Patterns

It is not always easy to construct a temporal logic formula which specifies precisely a given property, e.g. the specification of a property of periodicity with bounded jitter will be seen shortly to require some effort. This problem has received some attention with respect to the qualitative logics LTL and CTL, for which specification *patterns* have been identified for a variety of commonly required properties [DAC98]. It is possible to apply this approach also to quantitative logics like TCTL. We give here a small selection of some simple property patterns.

Invariance $\forall\Box\,\phi$ — ϕ is invariantly true, i.e., it holds in all states along all executions

Bounded Invariance $\forall\Box_I\,\phi$ — ϕ is satisfied continuously throughout the interval I.

Bounded Inevitability $\forall\Diamond_I\,\phi$ — ϕ is satisfied eventually at some time within the interval I;

Bounded Potentiality $\exists\Diamond_I\,\phi$ — ϕ is satisfied eventually at some time within the interval I, along at least one execution.

Upper Bounded Response $\forall\Box(\phi_1 \Rightarrow \forall\Diamond_{\leq t}\,\phi_2)$ — ϕ_2 is satisfied within at most t time units of the satisfaction of ϕ_1

Lower Bounded Response $\forall\Box(\phi_1 \Rightarrow \neg\;\exists\Diamond_{\leq t}\,\phi_2)$ — satisfaction of ϕ_2 is separated by at least t time units from the satisfaction of ϕ_1

Non-Zenoness $init \Rightarrow \forall\Box\,\exists\Diamond_{=1}\,\mathbf{true}$ — Assume that *init* uniquely characterises the initial state of a system. Then, the truth of this formula implies that the system is non-Zeno, i.e. that from any reachable state, time can progress without bound [HNSY94].

Periodicity with bounded jitter $\forall\Diamond\,\phi \;\wedge\; \forall\Box(\phi \;\Rightarrow\; \forall\Diamond_{\leq t}((\forall\Box_{<t_1} \neg\; \phi) \;\wedge\; (\forall\Diamond_{\leq t_2}\,\phi)))$ — Assume that ϕ stands for $\mathsf{enable}(a)$ which holds iff the action a is enabled. Assume also that a always occurs within t time units of becoming enabled. Then the formula above specifies that a occurs periodically, the distance between occurrences being in the interval $[t_1, t_2 + t]$.

There is a need for a more systematic approach to the development of property patterns for TCTL, with a view to developing a useful library.

Example 2.9 Consider again the level crossing controller of Figure 2.3. The safety property 'the gate is closed whenever the train is in the crossing' can be expressed in TCTL as $\mathit{init} \Rightarrow \forall\Box(\mathtt{Train}@2 \Rightarrow \mathtt{Gate}@2)$, and the bounded response property 'the gate is always opened within 10 seconds of being closed' as $\mathit{init} \Rightarrow \forall\Box(\mathtt{Gate}@2 \Rightarrow \forall\Diamond_{<10}\,\mathtt{Gate}@0)$. □

2.6.4 Discussion

Naturally enough, the literature on both timed and untimed formalisms is replete with discussions concerning the pros and cons of specification using automata and temporal logic, and of the relationship between them [AH91, BVW94, DW99, GPVW95, HKV96, Var96, VW86, VW94]. A prevalent view is that automata, because of their explicit structure and simple, operational semantics, are better suited to the construction of verification algorithms, while temporal logics, because of their concise, more readable syntax, are better suited to the expression of specifications. An obvious direction to follow in the search for practical and usable formal methods is to see to what extent it is possible to automate the translation of specifications expressed in temporal logic to equivalent automata which can be used for verification. In the case of the qualitative, linear-time logic LTL, this has been achieved [GPVW95] and found to lead to an efficient, on-the-fly model checking procedure which has been implemented in the verification tool SPIN [Hol96].

A similar relationship between the branching-time logic CTL and alternating tree automata has been established in [BVW94]. This work has been extended to TCTL in [HKV96] and the relationship between TCTL and timed alternating tree automata is further developed in [DW99]. Although, this work lays the theoretical foundations for efficient, on-the-fly model checking for TCTL, we know of no implementations of the ideas or experimental results which demonstrate their effectiveness in practice.

The relationship between temporal logic and testing automata is studied in [ABL98]. The authors introduce a restricted safety and bounded liveness logic (SBLL) and demonstrate that for any closed formula ϕ of SBLL and any TA $\mathcal{A}_M$, there is a test automaton $\mathcal{A}_S$ such that $\mathcal{A}_M$ satisfies ϕ iff no error state is reachable in $\mathcal{A}_M \mid \mathcal{A}_S$[3]. Moreover, they show how to construct $\mathcal{A}_S$

[3] The notions of satisfaction and parallel composition used in [ABL98] differ somewhat from those used in this dissertation, but their work is of interest and relevance, even so.

automatically from ϕ. In [ABBL98], a complete characterisation is provided of the class of properties of TA for which model-checking can be reduced to reachability analysis in the context of testing automata.

In conclusion, we remark that a variety of techniques are useful in property specification. Most often, specifications are more succinctly and clearly expressed with temporal logic than with automata. Even so, support in the form of a library of specification templates would be welcome. Restricting oneself to a logic such as SBLL allows for the automatic generation of test automata which can be used in model-checking based on efficient reachability techniques. However, the use of a logic such as TCTL permits the expression of a wider range of properties. Quite often, human ingenuity enables us to construct a test automaton or annotate an existing automaton in such a way that a verification problem can be solved more efficiently, but in adopting this approach, we need to be especially vigilant that we have really specified the property that was intended.

2.7 Verification

Verification is the conclusive demonstration that a system model possesses some well-specified property. It can take many forms, depending on the form of the model and the property. In this work, we are concerned primarily with *reachability analysis*. We assume that a system model is given as a TA $\mathcal{A}$ and that the property of interest is the reachability of some set of target states from a specified source state, along some time-divergent run in the transition system $\mathcal{T}[\![\mathcal{A}]\!]$. As we have seen, verification of safety properties of real-time systems can be formulated as reachability problems for TA. Also, the techniques developed in the solution of the reachability problem provide the basis for solutions to a wide variety of other verification problems such as model checking and language emptiness. The difficulty of the reachability problem for TA is caused by the infinite state spaces which inevitably arise because of the dense nature of the time domain. Solutions to the problem are based upon the identification of a finite number of classes of equivalent states which partition the infinite state space. We introduce the main ideas below.

2.7.1 Region Equivalence

The classic equivalence which is the foundation for most of the verification results on timed automata is the *region equivalence* [AD90, Alu91, ACD93, AD94]. Region equivalence has the crucial property of inducing a finite partition of the state space while preserving both linear time properties (such as reachability and TBA-emptiness) and branching time properties (such as TCTL satisfaction). Informally, clock valuations are region equivalent if they agree on the integral parts of all clock values and on the ordering of the fractional parts of all clock values. This idea on its own does not lead to a finite number of equivalence classes, since clock values can grow arbitrarily large. However once the value of a clock exceeds the largest constant c to which it is compared in a clock constraint, then its actual value is of no further interest – it is simply greater than c. These ideas, taken together, give the basis for a finite partitioning of the infinite space of clock valuations, which is presented formally below.

Definition 2.13 (Region Equivalence) Let $t \in \mathbb{R}$. We denote by $\lfloor t \rfloor$ the greatest integer smaller than or equal to t and by $\langle t \rangle$ the value $t - \lfloor t \rfloor$. Let $\mathcal{A}$ be a timed automaton with set of clocks $\mathcal{H} = \{h_1, h_2, \ldots, h_n\}$. For $i = 1, 2, \ldots, n$, let $c_i \geq c_{max}(\mathcal{A}, h_i)$. Two $\mathcal{H}$-valuations $\mathbf{v}$ and $\mathbf{v}'$ are *region equivalent*, denoted $\mathbf{v} \simeq \mathbf{v}'$, iff for $1 \leq i, j \leq n$ the following conditions hold:

1. $\mathbf{v}(h_i) > c_i$ iff $\mathbf{v}'(h_i) > c_i$
2. if $\mathbf{v}(h_i) \leq c_i$ then
   (a) $\lfloor \mathbf{v}(h_i) \rfloor = \lfloor \mathbf{v}'(h_i) \rfloor$
   (b) $\langle \mathbf{v}(h_i) \rangle = 0$ iff $\langle \mathbf{v}'(h_i) \rangle = 0$
   (c) $\langle \mathbf{v}(h_i) \rangle \leq \langle \mathbf{v}(h_j) \rangle$ iff $\langle \mathbf{v}'(h_i) \rangle \leq \langle \mathbf{v}'(h_j) \rangle$ □

It can be shown that $\simeq$ is an equivalence relation, whatever the values of c_i, and that it partitions $\mathbb{R}^{\mathcal{H}}$ into a finite number of equivalence classes, called *clock regions*. The clock region including $\mathbf{v}$ is denoted $[\mathbf{v}]$. A clock region of $\mathbb{R}^{\mathcal{H}}$ is known as a $\mathcal{H}$-region. A clock region ρ is said to be *unbounded* if for all $\mathbf{v} \in \rho$, $\mathbf{v}(h_i) > c_i$, for $i = 1, 2, \ldots, n$. Clearly, the values of all clocks in an unbounded region ρ may grow without bound and $[\mathbf{v} + t] = \rho$, for all $t \in \mathbb{R}$. It is a useful property of region equivalence that every clock region can be characterised uniquely by a clock constraint which it satisfies. When convenient, we will identify a clock region with the constraint which characterises it.

Example 2.10 Figure 2.6 shows an example of the region equivalence for two clocks h_1 and h_2 with maximal constants $c_1 = c_2 = 2$. Some characteristic constraints are shown. □

The number of clock regions is finite and bounded from above [ACD93] by

$$n! \cdot 2^n \cdot \prod_{i \leq n} (2 \cdot c_i + 2)$$

It can be shown that for any clock constraint ψ of $\mathcal{A}$, if $\mathbf{v} \simeq \mathbf{v}'$ then $\mathbf{v} \models \psi$ iff $\mathbf{v}' \models \psi$.

2.7.2 Region Graph

The region equivalence $\simeq$ over clock valuations can be extended to an equivalence relation over the state space of $\mathcal{A}$. Let $(\Sigma, \sigma^{\mathcal{I}}, L, \longrightarrow)$ be the transition system of $\mathcal{A}$. Two states from Σ are equivalent if they have identical locations and their clock valuations are region equivalent. Formally, for $(q, \mathbf{v}), (q', \mathbf{v}') \in \Sigma$, $(q, \mathbf{v}) \simeq (q', \mathbf{v}')$ iff $q = q'$ and $\mathbf{v} \simeq \mathbf{v}'$. The region (equivalence class) of $\sigma = (q, \mathbf{v})$ is denoted $[\sigma]$. The key property of region equivalence is its *stability* with respect to the transition relation of $\mathcal{A}$, stated as follows:

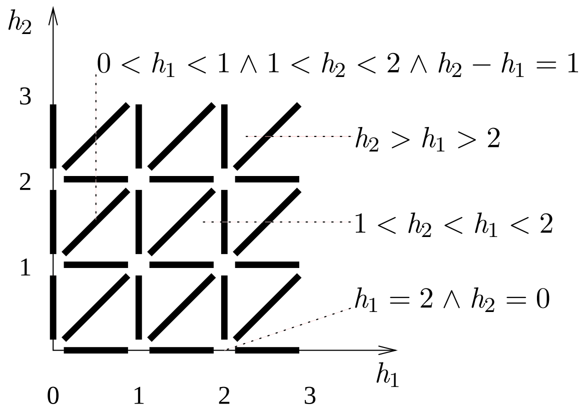


Fig. 2.6: Clock regions on $\{h_1, h_2\}$ with $c_1 = c_2 = 2$

Proposition 2.1 (Stability of region equivalence) *Let $\mathcal{T}[\![\mathcal{A}]\!] = (\Sigma, \sigma^{\mathcal{I}}, A \cup \mathbb{R}, \longrightarrow)$ be the transition system of $\mathcal{A}$. Let $\sigma_1 \simeq \sigma_2$.*

1. *For all $a \in A$, whenever $\sigma_1 \xrightarrow{a} \sigma_1'$, there exists σ_2' such that $\sigma_2 \xrightarrow{a} \sigma_2'$ and $\sigma_1' \simeq \sigma_2'$.*

2. *For all $t \in \mathbb{R}$, whenever $\sigma_1 \xrightarrow{t} \sigma_1'$, there exists σ_2' and $t' \in \mathbb{R}$ such that $\sigma_2 \xrightarrow{t'} \sigma_2'$, and $\sigma_1' \simeq \sigma_2'$.* □

We can gain an informal understanding of stability by considering again the regions of Figure 2.6. A state change can occur either through a discrete transition or a time transition. For a discrete transition, if two states are in the same region then they satisfy the same set of guards and so if the transition is possible for one state then it is also possible for the other. In taking the transition, one or more of the clocks h_1, h_2 may be set to 0. Assume that h_2 is reset. This give a projection onto the h_1 axis. It can be seen that equivalent states are projected to equivalent states. For a time transition, since both h_1 and h_2 increase at the same rate, the state change occurs along the diagonal at 45° to the h_1 axis. Again it can be seen that for any region and any pair of states within it, the sequence of regions encountered on the diagonal is the same.

Definition 2.14 (Region Graph [ACD93, Yov97]**)** Let $\mathcal{T}[\![\mathcal{A}]\!] = (\Sigma, \sigma^{\mathcal{I}}, A \cup \mathbb{R}, \longrightarrow)$. Let $\simeq$ be a region equivalence for $\mathcal{A}$ over Σ. Let $\tau \notin A$ and $A_\tau = A \cup \{\tau\}$. The *region graph* $\mathsf{RG}(\mathcal{A})$ is given by $(\Sigma_\simeq, [\sigma^{\mathcal{I}}], A_\tau, \longrightarrow_{\mathsf{rg}})$ where

1. $\Sigma_\simeq = \{[\sigma] \mid \sigma \in \Sigma\}$

2. $\longrightarrow_{\mathsf{rg}} \subseteq \Sigma_\simeq \times A_\tau \times \Sigma_\simeq$ is such that

   (a) for all $a \in A$ and for all $\rho, \rho' \in \Sigma_\simeq$, $\rho \xrightarrow{a}_{\mathsf{rg}} \rho'$ iff there exists $\sigma, \sigma' \in \Sigma$ such that $\rho = [\sigma]$, $\rho' = [\sigma']$, and $\sigma \xrightarrow{a} \sigma'$.

(b) for all $\rho, \rho' \in \Sigma_{\simeq}$, $\rho \xrightarrow{\tau}_{\mathsf{rg}} \rho'$ iff

i. $\rho = \rho'$ is an unbounded region, or

ii. $\rho \neq \rho'$ and there exists $\sigma, \sigma' \in \Sigma$ and $t \in \mathbb{R}$ such that $\sigma \xrightarrow{t} \sigma'$, and $\rho = [\sigma]$ and $\rho' = [\sigma']$, and for all $t' \in \mathbb{R}$, if $t' \leq t$ then $[\sigma + t]$ is either ρ or ρ'. □

In the region graph, the passage of time is indicated by the occurrence of a τ-transition which records the fact that time has passed but abstracts the exact amount of time elapsed. $\mathsf{RG}(\mathcal{A})$ is known as a *time-abstract* transition system.

From the stability of the region equivalence, it is clear that a state σ' is reachable from a state σ in the transition system of $\mathcal{A}$ iff $[\sigma']$ is reachable from $[\sigma]$ in the region graph of $\mathcal{A}$. It is also clear that $\mathsf{RG}(\mathcal{A})$ is finite since $\Sigma_{\simeq} = \{(q, [\mathbf{v}]) \mid q \in Q \wedge \mathbf{v} \in \mathbb{R}^{\mathcal{H}} \wedge \mathbf{v} \models I(q)\}$ is finite, A_τ is finite and therefore $\longrightarrow_{\mathsf{rg}} \subseteq \Sigma_{\simeq} \times A_\tau \times \Sigma_{\simeq}$ is finite. It follows that reachability can be decided automatically by constructing and searching the region graph. Both forward and backward traversals of the region graph lead to effective algorithms. For example, a method based on forward traversal consists in starting from $[\sigma]$ and visiting the set of its successors and the successors of those and so on, until all reachable regions have been visited. In this way, we construct the sequence $Z_0 \subseteq Z_1 \subseteq \cdots$, such that

$$\begin{aligned} Z_0 &= [\sigma] \\ Z_{i+1} &= Z_i \cup \{\rho \mid \exists \rho_i \in Z_i \,.\, \rho_i \longrightarrow_{\mathsf{rg}} \rho\} \end{aligned}$$

Assume that $Z = \lim_{i \geq 0} Z_i$. Then, $[\sigma']$ is reachable from $[\sigma]$ iff $[\sigma'] \in Z$.

2.7.3 Complexity of reachability

A timed automaton $\mathcal{A}$ with m locations and n clocks, in which $c \geq c_{max}(\mathcal{A})$, gives rise to a region graph with at most $m \cdot n! \cdot 2^n \cdot (2c+2)^n$ nodes. This bound is linear in the number of locations but exponential both in the number of clocks and the size of the constants appearing in the clock constraints. It can be shown that the number of edges in the region graph is similarly related to the number of locations and clocks and the size of constants [AD94]. In order to determine if a state σ is reachable in $\mathcal{T}[\![\mathcal{A}]\!]$, we search the region graph to see if $[\sigma]$ is reachable in $\mathsf{RG}(\mathcal{A})$ – Figure 2.7 outlines an algorithm to achieve this. Such a search is linear in the number of nodes and edges of the region graph. Therefore, the complexity of the reachability problem for $\mathcal{A}$ is linear in the number of locations, exponential in the number of clocks and exponential in the size of the constants in the clock constraints. Formally, the problem is shown to be PSPACE-complete [AD94]. In fact, it is usually the case, in practice, that $\mathcal{A}$ is a product of component automata, so the region graph can be seen as being exponential also in the number of component automata. To summarise the causes of complexity, we can identify the following factors:

1. the number of component automata

```
VISITED := {(q^I, [0])}
WAITING := {(q^I, [0])}
while WAITING ≠ ∅ do
  remove some ρ from WAITING
  succ := {ρ_s | ρ ⟶_rg ρ_s}
  foreach ρ_s ∈ succ do
    if ρ_s ∉ VISITED
      add ρ_s to VISITED
      add ρ_s to WAITING
    fi
  od
od
```

Fig. 2.7: Region graph reachability

2. the number of clocks
3. the size of the constants in the clock constraints

The combination of these factors cause a rapid growth in the number of states which must be considered, as the size of the problem description increases. This rapid growth is known as the *state explosion problem* and is currently the most challenging of the technical difficulties to be addressed in the application of automated analysis to formal verification problems in the analysis of real-time systems.

The state explosion problem

Consider again the algorithm for generating reachable regions in Figure 2.7. It can be seen that the algorithm stores each region from the region graph in the set *VISITED*. For the purposes of this algorithm, a 'state' is equated with a region. Storing the set of *VISITED* states makes termination of the algorithm easy to determine and ensures that states are not explored (have their successors generated) more than once. However, because the number of states can be very large, the available computational resources may become exhausted before the problem is solved. A number of attacks on the state explosion problem can be suggested:

1. generate fewer 'states',
2. store fewer 'states',
3. compress the 'state' store so that it requires less memory.

Such methods may be orthogonal and so can be combined to produce even greater benefits. In the following section, we consider one such approach which has proven successful in practice and is the basis for some of the most effective verification tools currently in use.

2.7.4 Constraint Solving

The partitioning of the space of clock valuations which arises in the construction of the region graph, although finite, is very fine-grained. Consequently, implementations based directly on the region graph turn out to be not very efficient. In [HNSY94] a symbolic technique was proposed which works directly with the clock constraints which arise in the calculation of discrete- and time-predecessors (and successors). This technique leads to a much coarser partitioning of the state space. The method of [HNSY94] works in a 'backward' manner, whereby starting from a set of target locations, the set of all states from which it is possible to reach those locations is calculated – it is then simple to test if an initial state lies within this set. In fact, this method is used in solving the model-checking problem for TCTL rather than simple reachability. The main problems with a backward traversal of the state space are:

- the whole of the potential state space may be considered rather than just that part which is reachable from an initial state;
- an answer cannot be returned until the complete state space exploration terminates;
- it is not easy to provide a diagnostic trace in the case that a violating state is found to be reachable.

The idea of working symbolically with clock constraints in a 'forward' manner seems to have arisen independently, at about the same time, in several groups [ACD$^+$92, Oli94, YPD94]. This approach is often more efficient in practice, allows for a diagnostic trace to be provided when a property is found to be violated and is the basis of successful implementations [BLL$^+$95, DOTY95]. We rely on 'forward' constraint solving techniques in Chapter 5 and provide an introduction below.

Symbolic states

A node in the region graph of a TA $\mathcal{A}$ is a 'symbolic' state which represents a (possibly infinite) number of states in the transition system of $\mathcal{A}$. Each node is of the form (q, ρ) where q is a location of $\mathcal{A}$ and ρ is a clock region. Such a symbolic state represents the set of states $(q, \mathbf{v})$ where $\mathbf{v} \in \rho$. We have seen that every clock region can be characterised by a clock constraint, so a node (q, ρ) can be written as (q, ψ) where ψ is the characteristic formula of the region ρ. This idea can be extended by allowing ψ to be a constraint which characterises a *union* of perhaps many clock regions. Formally, a symbolic state is defined as follows.

Definition 2.15 (Symbolic state) Let $\mathcal{A} = (Q, q^{\mathcal{I}}, A, \mathcal{H}, E, I)$ be a timed automaton. A *symbolic state* of $\mathcal{A}$ is a pair (q, ψ) where $q \in Q$ is a location of $\mathcal{A}$ and $\psi \in \Psi_{\mathcal{H}}$ is a clock constraint.

The meaning of a symbolic state (q, ψ), denoted $[\![(q, \psi)]\!]$, is the set of states $\{(q, \mathbf{v}) \mid \mathbf{v} \models \psi\}$. □

Let Z be a set of symbolic states. We denote by $[\![Z]\!]$ the set $\bigcup\{[\![(q,\psi)]\!] \mid (q,\psi) \in Z\}$ and by $\mathsf{locations}(Z)$ the set of locations $\{q \mid \exists\, \psi \in \Psi_{\mathcal{H}} \,.\, (q,\psi) \in Z\}$.

The state space may be covered by a much smaller set of symbolic states of this form, and so the problem of state space explosion may be mitigated to some extent. In particular, a set of regions Z can be represented as a *united* set of symbolic states $Z' = \{(q,\psi_q) \mid q \in Q \wedge \psi_q \in \Psi_{\mathcal{H}}\}$ in which there is at most one element (q,ψ_q) for each location q, and ψ_q is the characteristic formula for the set of *all* clock regions in Z which are paired with the location q.

Let Z be a set of symbolic states. We denote by ψ_q^Z the clock constraint characterising the set of clock valuations associated with q in Z, i.e., $[\![\psi_q^Z]\!] = \bigcup\{[\![\psi]\!] \mid (q,\psi) \in Z\}$. We use $\mathsf{unite}(Z)$ to denote the set $\{(q,\psi_q^Z) \mid q \in \mathsf{locations}(Z)\}$, and $Z_1 \uplus Z_2$ to denote $\mathsf{unite}(Z_1 \cup Z_2)$ and $\biguplus_{i\in I} Z_i$ to denote $\mathsf{unite}(\bigcup_{i\in I} Z_i)$.

In the following section, we discuss the calculation of the discrete and time-successors of symbolic states, and show how united sets of symbolic states can be used in the forward computation of reachable states.

Forward computation of clock constraints

Let $q \in Q$, $\psi \in \Psi_{\mathcal{H}}$ and $e = (q,\zeta,a,\mathsf{H},q') \in E$. We consider predicate transformers $\mathsf{suc}_e(\psi)$ and $\mathsf{suc}_\tau^q(\psi)$ which are needed in the calculation of discrete and time-successors, respectively, of a symbolic state (q,ψ).

On the one hand, $\mathsf{suc}_e(\psi)$ denotes a clock constraint over $\mathcal{H}$ which characterises the set of clock valuations which are reachable from the clock valuations in ψ when a discrete transition is taken via the edge e, i.e., $\mathsf{suc}_e(\psi)$ denotes a predicate satisfying

$$[\![\,\mathsf{suc}_e(\psi)]\!] = \{\mathbf{v}[\mathsf{H} := 0] \mid \mathbf{v} \in \mathbb{R}^{\mathcal{H}} \wedge (\mathbf{v} \models \psi \wedge \zeta) \wedge \mathbf{v}[\mathsf{H} := 0] \models I(q')\}$$

On the other hand, $\mathsf{suc}_\tau^q(\psi)$ denotes a clock constraint over $\mathcal{H}$ which characterises the set of clock valuations which are reachable from the clock valuations in ψ as time passes while control resides at q, i.e., $\mathsf{suc}_\tau^q(\psi)$ denotes a predicate satisfying

$$\begin{aligned}[\![\,\mathsf{suc}_\tau^q(\psi)]\!] = \{\mathbf{v} + t \mid {} & \mathbf{v} \in \mathbb{R}^{\mathcal{H}} \wedge t \in \mathbb{R} \wedge \mathbf{v} \models \psi \wedge \\ & \forall\, t' \in \mathbb{R} \,.\, t' \leq t \Rightarrow \mathbf{v} + t' \models I(q)\}\end{aligned}$$

Together, $\mathsf{suc}_e(\psi)$ and $\mathsf{suc}_\tau^q(\psi)$ can be used in solving the reachability problem by computing the sequence of sets of symbolic states $Z_0, Z_1, \cdots$ as follows:

$$\begin{aligned} Z_0 &= \{(q,\psi)\} \\ Z_{i+1} &= \{(q',\mathsf{suc}_e(\psi)) \mid (q,\psi) \in Z_i \wedge e = (q,\zeta,a,\mathsf{H},q') \in E\} \uplus \\ & \{(q,\mathsf{suc}_\tau^q(\psi)) \mid (q,\psi) \in Z_i\} \end{aligned}$$

Notice that $Z_i \subseteq \{(q,\mathsf{suc}_\tau^q(\psi)) \mid (q,\psi) \in Z_i\}$. Let $Z = \lim_{i\geq 0} Z_i$. All states in a symbolic state (q',ψ') are reachable from some state in (q,ψ) iff $(q',\psi'') \in Z$ and $[\![\psi']\!] \subseteq [\![\psi'']\!]$, i.e., ψ' implies ψ''.

Implementing the constraint solving approach

In order to exploit these ideas in practice, it is necessary to see how it is possible to represent clock constraints and to implement the operations $\uplus$, suc_e and suc_τ using this representation. As a first step, we observe that $\Psi_{\mathcal{H}}$ is closed under these operations for any timed automaton, i.e., the operations are always well-defined – the reader is referred to [Oli94] for a proof. Next, we note that the adoption of a 'geometric' perspective leads to natural definitions of many of the operations which are needed and helps in acquiring an intuitive understanding of them. We follow this approach below.

Polyhedra

Let $\mathcal{H} = \{h_1, h_2, \ldots, h_n\}$ be a set of clocks. A union of $\mathcal{H}$-regions is a $\mathcal{H}$-*polyhedron* in the n-dimensional Euclidean space, where a $\mathcal{H}$-polyhedron is simply the set of $\mathcal{H}$-valuations satisfying a clock constraint $\psi \in \Psi_{\mathcal{H}}$. It is often convenient, notationally, to identify a constraint with the $\mathcal{H}$-polyhedron which it defines, and so, for example, we will write $\mathbf{v} \in \psi$ for $\mathbf{v} \in [\![\psi]\!]$, or $\psi_1 \cup \psi_2$ for $[\![\psi_1]\!] \cup [\![\psi_2]\!]$.

A polyhedron is said to be *convex*, if for any two points within it, all points on the line segment joining them are also within it. Formally, a $\mathcal{H}$-polyhedron ζ is convex iff for any $\mathbf{v}_1, \mathbf{v}_2 \in \zeta$ and $t \in \mathbb{R}$ such that $0 < t < 1$, we have $t \cdot \mathbf{v}_1 + (1-t) \cdot \mathbf{v}_2 \in \zeta$. This means that if $\mathbf{v}$ and $\mathbf{v} + t$ are clock valuations, both of which lie within a convex polyhedron ψ, then all valuations $\mathbf{v} + t'$, where $t' \leq t$, also lie within ψ.

It can be shown that the set of convex $\mathcal{H}$-polyhedra coincides with the set $\mathcal{Z}_{\mathcal{H}}$ of clock zones, i.e., any convex $\mathcal{H}$-polyhedron can be expressed as a conjunction of atomic constraints, and any conjunction of atomic constraints defines a convex $\mathcal{H}$-polyhedron. Note that any non-convex $\mathcal{H}$-polyhedron, ψ, can be expressed as the union of a finite set of convex $\mathcal{H}$-polyhedra, $\bigcup\{\zeta_1, \zeta_2, \ldots, \zeta_m\}$.

Example 2.11 Figure 2.8 shows (a) one convex and (b,c) two non-convex polyhedra, which are unions of clock regions and are defined by the constraints:

a) $1 \leq h_1 \leq 3 \wedge 1 \leq h_2 \leq 3 \wedge -1 \leq h_2 - h_1 \leq 1$

b) $(0 \leq h_1 \leq 3 \wedge 0 \leq h_2 \leq 1 \wedge 0 \leq h_1 - h_2 \leq 2) \vee$
$(1 \leq h_1 \leq 2 \wedge 2 \leq h_2 \leq 3)$

c) $(0 \leq h_1 \leq 1 \wedge 1 \leq h_2 \leq 3) \vee (1 \leq h_1 \leq 2 \wedge 1 \leq h_2 \leq 2)$ □

Operations on polyhedra

In this section, we define a number of operations on polyhedra which are needed in the rest of the dissertation. Some of the operations are illustrated in Figure 2.9 where the result of each operation is indicated by the shaded part in each case.

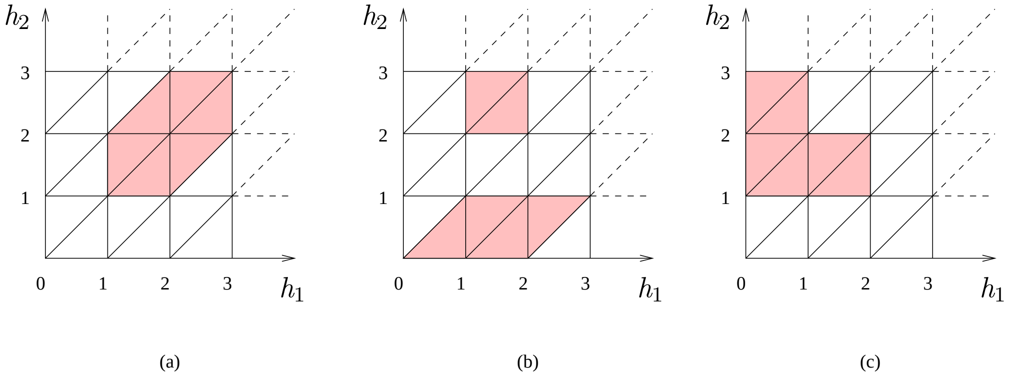


Fig. 2.8: Convex and Non-convex Polyhedra

Basic operations *Intersection*, *union* and *complementation* are given immediately by conjunction, disjunction and negation respectively, i.e., $\psi_1 \cap \psi_2 = \{\mathbf{v} \in \mathbb{R}^{\mathcal{H}} \mid \mathbf{v} \models \psi_1 \wedge \psi_2\}$, $\psi_1 \cup \psi_2 = \{\mathbf{v} \in \mathbb{R}^{\mathcal{H}} \mid \mathbf{v} \models \psi_1 \vee \psi_2\}$ and $\overline{\psi} = \{\mathbf{v} \in \mathbb{R}^{\mathcal{H}} \mid \mathbf{v} \models \neg\, \psi\}$ – examples of intersection and union are given in Figures 2.9(a) and (b), respectively. *Difference* is defined as usual by $\psi_1 \setminus \psi_2 = \psi_1 \cap \overline{\psi_2}$ and the *inclusion* $\psi_1 \subseteq \psi_2$ is equivalent to $\psi_1 \setminus \psi_2 = \emptyset$.

Convex hull The *convex hull* of two $\mathcal{H}$-polyhedra ψ_1 and ψ_2 is denoted $\psi_1 \sqcup \psi_2$, and is defined to be the smallest convex $\mathcal{H}$-polyhedron ζ which contains both ψ_1 and ψ_2, i.e., $\psi_1 \subseteq \zeta$ and $\psi_2 \subseteq \zeta$. Figure 2.9(c) gives an example of the convex hull operation.

Projections The *forward projection* of a $\mathcal{H}$-polyhedron ψ, denoted $\nearrow\psi$, is the largest set of $\mathcal{H}$-valuations which can be obtained from the valuations in ψ by the passage of time. Formally,

$$\nearrow\psi \mathrel{\widehat{=}} \{\mathbf{v} + t \mid \mathbf{v} \in \psi \wedge t \in \mathbb{R}\}$$

For a polyhedron ψ on $\{h_1, h_2\}$, since h_1 and h_2 advance together in lock-step with the passage of time, the forward projection $\nearrow\psi$ encompasses all those valuations which can be reached from a valuation in ψ by following the diagonal at 45° to the horizontal axis. Figure 2.9(d) shows an example.

The operation giving the *reset successors* of a $\mathcal{H}$-polyhedron ψ, for a given reset set $\mathsf{H} \subseteq \mathcal{H}$, is denoted $\psi[\mathsf{H} := 0]$ and is defined by:

$$\psi[\mathsf{H} := 0] \mathrel{\widehat{=}} \{\mathbf{v}[\mathsf{H} := 0] \mid \mathbf{v} \in \psi\}$$

Intuitively, the reset of a clock h_2, for a polyhedron ψ, involves a projection of ψ onto the h_1 axis – see Figure 2.9(e).

c-closure The operation of c-closure, defined on convex polyhedra, is based on the idea that if the value of some clock exceeds a specified constant c in each of two clock valuations, then that clock is not regarded as significant in distinguishing between them. The c-closure operation is used to ensure the finiteness

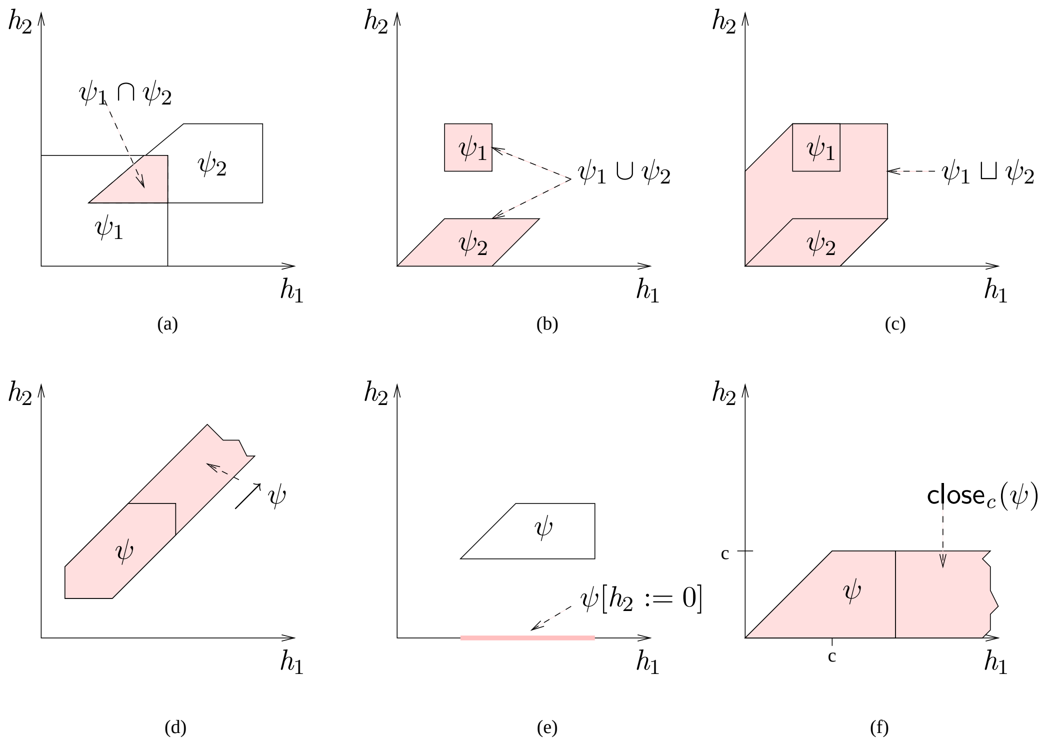


Fig. 2.9: Operations on Polyhedra

of partitionings of the infinite space of clock valuations. We have seen a similar idea already in connection with the region graph (§2.7.2). c-closure appears in the literature under a variety of names, e.g., *rounding* [Won95], *extrapolation* [DT98] and *normalisation* [Pet99]. The definition given here follows [Tri98].

Let $c \in \mathbb{N}$ and $\mathbf{v}, \mathbf{v}' \in \mathbb{R}^{\mathcal{H}}$. We say that $\mathbf{v}$ and $\mathbf{v}'$ are *c-equivalent* if:

1. for any clock h, either $\mathbf{v}(h) = \mathbf{v}'(h)$, or $\mathbf{v}(h) > c$ and $\mathbf{v}'(h) > c$, and
2. for any pair of clocks, h_1, h_2, either $\mathbf{v}(h_1) - \mathbf{v}(h_2) = \mathbf{v}'(h_1) - \mathbf{v}'(h_2)$, or $\mathbf{v}(h_1) - \mathbf{v}(h_2) > c$ and $\mathbf{v}'(h_1) - \mathbf{v}'(h_2) > c$.

For a convex $\mathcal{H}$-polyhedron ζ, the c-closure of ζ, denoted $\mathsf{close}_c(\zeta)$, is defined to be the greatest convex $\mathcal{H}$-polyhedron $\zeta' \supseteq \zeta$, such that for all $\mathbf{v}' \in \zeta'$ there exists $\mathbf{v} \in \zeta$ and $\mathbf{v}$, $\mathbf{v}'$ are c-equivalent. ζ is said to be c-closed if $\mathsf{close}_c(\zeta) = \zeta$. Figure 2.9(f) shows an example of c-closure.

Proposition 2.2 *c-closure satisfies the following properties:*

1. *If ζ is c-closed then it is c'-closed, for any $c' \geq c$.*
2. *If ζ_1 and ζ_2 are c-closed then $\zeta_1 \cap \zeta_2$ is also c-closed.*
3. *For any ζ, there exists a constant c such that ζ is c-closed.*
4. *For any constant c, there is a finite number of c-closed convex $\mathcal{H}$-polyhedra.*

Proof cf. Tripakis [Tri98] □

Properties of polyhedral operations

Firstly, we identify those operations of the previous section which preserve convexity.

Proposition 2.3 *Let ζ, ζ_1, ζ_2 be convex $\mathcal{H}$-polyhedra. Let $\mathsf{H} \subseteq \mathcal{H}$ and $c \in \mathbb{N}$. Then, $\zeta_1 \cap \zeta_2$, $\zeta_1 \sqcup \zeta_2$, $\nearrow \zeta$, $\zeta[\mathsf{H} := 0]$ and $\mathsf{close}_c(\zeta)$ are all convex.*

Proof cf. Tripakis [Tri98] □

Proposition 2.4 *Let $\mathcal{A}$ be a timed automaton with a set $\mathcal{H}$ of clocks and a set E of edges with $e = (q, \zeta, a, \mathsf{H}, q') \in E$. Let ψ be a $\mathcal{H}$-polyhedron. The following equalities hold.*

$$\begin{aligned} \mathsf{suc}_e(\psi) &= ((\psi \cap \zeta)[\mathsf{H} := 0]) \cap I(q') \\ \mathsf{suc}_\tau^q(\psi) &= \nearrow \psi \cap I(q) \end{aligned}$$

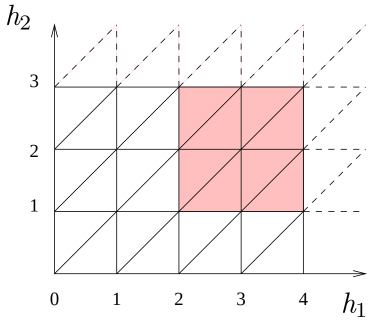
Proof The equalities can be derived directly from the definitions of suc_τ, suc_e and the polyhedral operations. □

Proposition 2.4 leads some way towards an implementation of the constraint-solving approach. In order to make further progress, we need to define an efficient representation for clock constraints and show how the polyhedral operations can be implemented on it. It is also necessary to consider the implications of the use of the $\uplus$ operator, which, in the general case, gives rise to non-convex polyhedra. The issues raised by this consideration are more easily discussed following the introduction of the *difference bound matrix* representation of clock constraints which is presented in the following section.

2.7.5 Difference Bound Matrices

The efficient implementation of algorithms for automatic analysis based on constraint solving relies upon a representation of polyhedra which is compact and which supports the operations identified in section 2.7.4. Dill [Dil89] introduced *difference bound matrices* (DBMs) for this purpose[4] and this data structure remains pre-eminent in the implementation of analysis tools for dense-time systems – KRONOS and UPPAAL are examples. We now present those details of DBMs and their use which will be needed later in the dissertation; more detailed presentations, including proofs, can be found in [Dil89, Oli94, Tri98, Yov93, Yov97].

[4] In fact, the data structure was known many years earlier [Bel57] and later had been used in the analysis of Time Petri nets [MB83] but Dill's paper revived interest and pointed the way to their use in the analysis of timed automata.



M	h_0	h_1	h_2
h_0	$(0, \leq)$	$(-2, \leq)$	$(-1, \leq)$
h_1	$(4, \leq)$	$(0, \leq)$	$(\infty, <)$
h_2	$(3, \leq)$	$(\infty, <)$	$(0, \leq)$

M'	h_0	h_1	h_2
h_0	$(0, \leq)$	$(-2, \leq)$	$(-1, \leq)$
h_1	$(4, \leq)$	$(0, \leq)$	$(3, \leq)$
h_2	$(3, \leq)$	$(1, \leq)$	$(0, \leq)$

Fig. 2.10: Representation of a convex polyhedron by DBM's

Bounds

A *bound* is a pair $(c, \prec) \in \mathbb{Z}_\infty \times \{<, \leq\}$, where $\mathbb{Z}_\infty = \mathbb{Z} \cup \{\infty\}$. Bounds are ordered as follows: $c < \infty$, for any $c \in \mathbb{Z}$, and $<$ is strictly less than $\leq$; we then take the usual lexicographic ordering where for all $(c, \prec), (c', \prec') \in \mathbb{Z}_\infty \times \{<, \leq\}$, $(c, \prec) < (c', \prec')$ if either $c < c'$, or $c = c'$ and $\prec < \prec'$. $(c, \prec) \leq (c', \prec')$ if $(c, \prec) < (c', \prec')$ or $c = c'$ and $\prec = \prec'$.

The *minimum* of two bounds $(c, \prec), (c', \prec')$, denoted $\mathsf{min}((c, \prec), (c', \prec'))$, is $(c, \prec)$ if $(c, \prec) \leq (c', \prec')$ and $(c', \prec')$ otherwise. The *maximum* of two bounds $(c, \prec), (c', \prec')$, denoted $\mathsf{max}((c, \prec), (c', \prec'))$, is $(c, \prec)$ if $(c', \prec') \leq (c, \prec)$ and $(c', \prec')$ otherwise. The *addition* of bounds is defined by the following table:

$+$	$(c', \leq)$	$(c', <)$
$(c, \leq)$	$(c + c', \leq)$	$(c + c', <)$
$(c, <)$	$(c + c', <)$	$(c + c', <)$

Note that as usual $c + \infty = \infty + c = \infty$ for any $c \in \mathbb{Z}_\infty$.

Representation of convex polyhedra

Let $\mathcal{H} = \{h_1, h_2, \ldots, h_n\}$ be a set of clocks. The set $\mathcal{Z}_\mathcal{H}$ of convex $\mathcal{H}$-polyhedra contains elements which are given as the conjunction of atomic constraints. An atomic constraint of the form $h_i - h_j \prec c$ can be represented by associating the bound $(c, \prec)$ with the pair of clocks h_i, h_j. A constraint such as $h_i - h_j \geq c$ is equivalent to $h_j - h_i \leq -c$ and so can be represented by associating the bound $(-c, \leq)$ with h_j, h_i. In order to achieve a uniform representation, a new fictitious clock variable h_0 is introduced to represent the constant 0. This allows constraints such as $h_i \prec c$ to be represented as $h_i - h_0 \prec c$. In this way, a convex $\mathcal{H}$-polyhedron can be encoded as a $(n+1) \times (n+1)$ square matrix M whose elements are bounds. Such a matrix is said to have *dimension* n. The element $M_{i,j}$ gives the upper bound on the clock difference $h_i - h_j$. For example, the constraint $h_2 < 9$ is encoded as $M_{2,0} = (9, <)$ and $h_5 \geq 6$ by $M_{0,5} = (-6, \leq)$. If $h_i - h_j$ is unbounded then we set $M_{i,j} = (\infty, <)$. The set of $\mathcal{H}$-valuations defined by the DBM M, denoted $[\![M]\!]$, is the set $\{\mathbf{v} \in \mathbb{R}^\mathcal{H} \mid \forall\, i, j \in \{0..n\}\ .\ M_{i,j} = (c, \prec) \Rightarrow \mathbf{v}(h_i) - \mathbf{v}(h_j) \prec c\}$. Notice that we silently extend $\mathbf{v}$ by requiring $\mathbf{v}(h_0) = 0$.

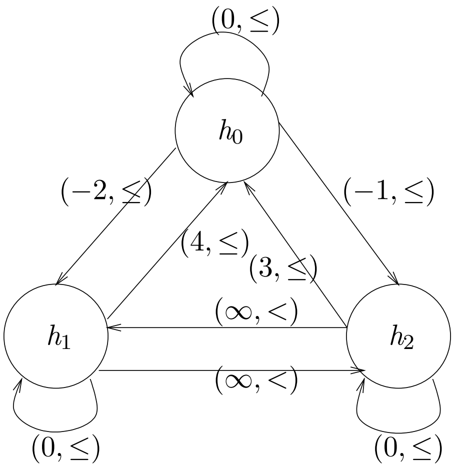


Fig. 2.11: Weighted graph interpretation of a DBM

Example 2.12 Let $\zeta = 2 \leq h_1 \leq 4 \wedge 1 \leq h_2 \leq 3$ be a clock constraint. Figure 2.10 illustrates the convex polyhedron defined by ζ and the DBM M which represents it. □

A DBM can also be regarded as the adjacency matrix of a fully connected, weighted directed graph, where each clock is a node in the graph and each entry $M_{i,j}$ gives the weight on the arc from h_i to h_j. Figure 2.11 shows the weighted graph corresponding to DBM M in Figure 2.10. We will use this interpretation whenever it is convenient in a given context.

Notice that there may be many DBMs which represent a given convex polyhedron, i.e., the representation is not unique. This can be observed in Figure 2.10 where M' represents the same polyhedron as M. A canonical representation is desirable since it allows certain semantic operations on polyhedra – the testing of equality, emptiness and inclusion, for example – to be reduced to syntactic operations on DBMs. An ordering on DBMs is induced by the ordering on bounds: $M \leq M'$ iff $M_{i,j} \leq M'_{i,j}$, for all $0 \leq i, j \leq n$ where n is the dimension of M and M'. This ordering allows a *canonical form* M_ζ to be defined for any non-empty convex polyhedron ζ: we require that $M_\zeta \leq M$, for any DBM M representing ζ, i.e., in the canonical form, all bounds are as 'tight' as possible. The empty polyhedron is defined by any inconsistent set of constraints. We choose its canonical form arbitrarily to be one of the many possible representations, denoting it $M^\emptyset$, where $M^\emptyset_{i,j} \mathrel{\widehat{=}} (0, <)$, for $0 \leq i, j \leq n$. If M is a DBM, then $\mathsf{cf}(M)$ denotes the canonical form of M and $[\![\mathsf{cf}(M)]\!] = [\![M]\!]$. In Figure 2.10, $M' = \mathsf{cf}(M)$. It is now simple to test if two matrices represent the same constraint: M and M' represent the same constraint if $\mathsf{cf}(M) = \mathsf{cf}(M')$.

Notice that $[\![M^\emptyset]\!] = \emptyset$ and so represents the constraint ff. The *universal* matrix $\mathbf{U}$, which imposes the minimal constraints that clock differences should be at least 0 and less than ∞, is defined by: $\mathbf{U}_{i,j} = (0, \leq)$ if $i = 0$ or $i = j$ otherwise $\mathbf{U}_{i,j} = (\infty, <)$. $[\![\mathbf{U}]\!] = \mathbb{R}^{\mathcal{H}}$ and so represents the constraint tt.

```
mk_canonical(M)
begin
  for k = 0 to n do
     for i = 0 to n do
        for j = 0 to n do
           M_{i,j} := min(M_{i,j}, M_{i,k} + M_{k,j})
        od
        if M_{i,i} < (0, ≤) then return M^∅ fi
     od
  od
  return M
end
```

Fig. 2.12: Procedure to compute the canonical form of a DBM

Implementation of polyhedra operations

Canonical Form The canonical form M', of a DBM M of dimension n, can be computed from the interpretation of M as a weighted directed graph by requiring that, for all $0 \leq i, j \leq n$, the weight $M'_{i,j} = \mathsf{min}\{\mathsf{weight}_M(\mathbf{p}) \mid \mathbf{p}$ is a path from h_i to $h_j\}$, where a path $\mathbf{p}$ from h_i to h_j is any sequence of nodes $h_i = h_{i_1}, h_{i_2}, \ldots, h_{i_m} = h_j$ and its weight in M, denoted $\mathsf{weight}_M(\mathbf{p})$, is given by $M_{i_1,i_2} + M_{i_2,i_3} + \cdots + M_{i_{m-1},i_m}$. If there is a cycle $h_i = h_{i_1}, h_{i_2}, \ldots, h_{i_m} = h_i$, such that $\mathsf{weight}_M(h_{i_1}, h_{i_2}, \ldots, h_{i_m}) < (0, \leq)$, then M represents the empty polyhedron — clearly, it cannot be the case that $h_i - h_i < 0$ — and its canonical form is $M^\emptyset$, otherwise the canonical form of M is given by M'. We can calculate the canonical form of a DBM by using a version of the Floyd-Warshall all-pairs shortest path algorithm, as shown in Figure 2.12. It is apparent that the complexity of the algorithm is $O((n+1)^3)$ for a DBM of dimension n.

Intersection Given two DBMs M and M' of dimension n, representing the convex polyhedra ζ, ζ', respectively, then the intersection $\zeta \cap \zeta'$ is represented by the DBM M'', where $M''_{i,j} = \mathsf{min}(M_{i,j}, M'_{i,j})$ for $0 \leq i, j \leq n$. This is true even if M and M' are not in canonical form. However, M'' is not necessarily in canonical form even if both M and M' are.

Inclusion Let M and M' be the DBMs of dimension n which are the canonical representatives of the convex polyhedra ζ and ζ', respectively. $\zeta \subseteq \zeta'$ iff $M_{i,j} \leq M'_{i,j}$, for $0 \leq i, j \leq n$.

Convex hull Let the DBMs M and M' of dimension n be the canonical representatives of the convex $\mathcal{H}$-polyhedra ζ and ζ', respectively. The DBM M'' given by $M''_{i,j} = \mathsf{max}(M_{i,j}, M'_{i,j})$, for $0 \leq i, j \leq n$, is the canonical representative of the convex $\mathcal{H}$-polyhedron $\zeta'' = \zeta \sqcup \zeta'$. If M and M' are not in canonical form, M'' still represents a convex polyhedron containing those represented by M and M', but it may not be the smallest one and it may not be in canonical form.

Projections Let ζ be a convex $\mathcal{H}$-polyhedron and let M be a DBM which represents ζ.

In the *forward projection*, $\nearrow\zeta$, which models the elapse of time, all clock differences remain the same, since all clocks increase at the same rate; lower bounds also remain unchanged, since clock values never decrease; however all upper bounds are removed, since time can advance beyond any bound. Therefore, if M is in canonical form, then M' is the canonical DBM representing $\nearrow\zeta$, where for $0 \leq i, j \leq n$:

$$M'_{i,j} = \begin{cases} (\infty, <), & \text{if } i > 0 \wedge j = 0 \\ M_{i,j}, & \text{otherwise} \end{cases}$$

If M is not in canonical form, then M' represents a superset of the forward projection.

The operation giving the DBM M', representing the *reset successors* $\zeta[\mathsf{H} := 0]$ of the polyhedron ζ, for the set of clocks $\mathsf{H} \subseteq \mathcal{H}$, is computed quite simply. First, notice that resetting a single clock $h_i \in \mathsf{H}$ is the same as setting the value of h_i to the value of h_0. So, all constraints on h_0 in M become constraints on h_i in M'. If a pair of clocks h_i, h_j are both reset, then clearly the differences $h_i - h_j$ and $h_j - h_i$ become equal to 0. Finally, if neither of a pair of clocks h_i, h_j is reset then the differences $h_i - h_j$ and $h_j - h_i$ remain unchanged in M'. Formally, if M is the canonical representative of ζ, then M' is the canonical representative of $\zeta[\mathsf{H} := 0]$, where for $0 \leq i, j \leq n$, the entry for $M'_{i,j}$ satisfies the following:

$$M'_{i,j} = \begin{cases} (0, \leq), & \text{if } h_i \in \mathsf{H} \wedge h_j \in \mathsf{H} \\ M_{0,j}, & \text{if } h_i \in \mathsf{H} \wedge h_j \notin \mathsf{H} \\ M_{i,0}, & \text{if } h_i \notin \mathsf{H} \wedge h_j \in \mathsf{H} \\ M_{i,j}, & \text{if } h_i \notin \mathsf{H} \wedge h_j \notin \mathsf{H} \end{cases}$$

If M is not in canonical form, then M' represents some convex $\mathcal{H}$-polyhedron $\zeta' \supseteq \zeta[\mathsf{H} := 0]$.

c-closure Given the canonical DBM M representing a polyhedron ζ, the c-closure of ζ, $\mathsf{close}_c(\zeta)$, is canonically represented by the DBM M', where, for $0 \leq i, j \leq n$:

$$M'_{i,j} = \begin{cases} (\infty, <), & \text{if } M_{i,j} > (c, \leq) \wedge i \neq j \\ (-c, <), & \text{if } M_{i,j} + (c, \leq) < (0, \leq) \wedge i \neq j \\ M_{i,j}, & \text{otherwise} \end{cases}$$

That is, an upper bound such as $h \leq c'$, where $c' > c$, is replaced by $h < \infty$. Also, a lower bound such as $h \geq c'$, where $c' > c$, is replaced by $h > c$. All other bounds remain unchanged.

Union and Complementation Clearly, $\mathcal{Z}_{\mathcal{H}}$ is not closed under union; this can be seen easily in Figure 2.9(b) which shows two convex polyhedra whose union is obviously non-convex. Similarly, complementation does not preserve

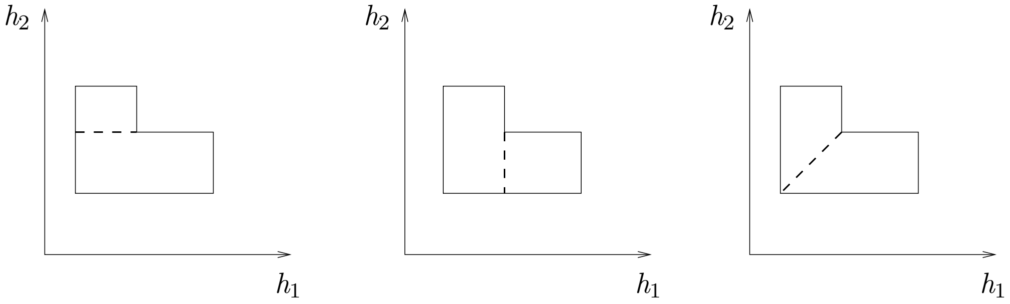


Fig. 2.13: Convex decompositions of a non-convex polyhedron

convexity. However, as we have observed, any non-convex polyhedron ψ can be expressed as a finite union $\bigcup\{\zeta_1, \zeta_2, \ldots, \zeta_m\}$ of convex polyhedra. This means that we can represent ψ as the set $\{M^1, M^2, \ldots, M^m\}$ where each M^i is the DBM encoding ζ_i. The representation of non-convex polyhedra as sets of DBMs has been implemented in tools such as KRONOS. It has been found that some polyhedral operations, projections for example, can still be implemented efficiently, but that others, such as intersection, are more expensive. A major problem, however, is that, in general, there is no obvious canonical form for a non-convex polyhedron. This is apparent in Figure 2.13 which shows a non-convex polyhedron and three of the possible ways in which it can be decomposed into convex polyhedra [Tri98]. It is not clear which, if any, of the decompositions is the most suitable canonical representative. The lack of a canonical form militates against the efficient testing of inclusion and equality. It is also difficult to check whether the union of two or more polyhedra is in fact convex, and so could be represented using a single DBM in order to reduce storage requirements.

2.7.6 Implementing constraint solving

Avoiding non-convex polyhedra

In the previous section, we have seen a number of pragmatic reasons for avoiding the use of non-convex polyhedra in implementing a constraint-solving approach to the reachability problem. This has motivated the investigation of methods which rely exclusively on convex polyhedra. Recall that the reachability problem can be solved by computing the limit of the sequence $Z_0, Z_1, \ldots$, where

$$\begin{aligned} Z_0 &= \{(q, \psi)\} \\ Z_{i+1} &= \{(q', \mathsf{suc}_e(\psi)) \mid (q, \psi) \in Z_i \wedge e = (q, \zeta_e, a, \mathsf{H}, q') \in E\} \uplus \\ &\quad\ \{(q, \mathsf{suc}^q_\tau(\psi)) \mid (q, \psi) \in Z_i\} \end{aligned}$$

It has been shown already that suc_τ and suc_e preserve convexity. However, $\uplus$ can give rise to non-convex polyhedra, because of the union of clock zones which is implicit in its definition. This union can be avoided simply by replacing it with a convex hull. In order to do this, we redefine ψ^Z_q, so that, for a set of symbolic states Z, $[\![\psi_q^Z]\!] = \bigsqcup\{\psi \mid (q, \psi) \in Z\}$. If we modify the definitions of unite and $\uplus$ to make use of this new definition, then all operations required in computing $Z_0, Z_1, \ldots$, preserve the convexity of polyhedra and so every clock constraint can be represented by a single DBM, with all of the efficiency gains which that implies. This approach has been adopted directly by Balarin [Bal96] who combines it with a representation of the complete state space using BDDs. The main problem with this method is that the convex hull gives only an (over-)approximation of the set of clock valuations associated with any location, and so, while the set of reachable states is clearly included in $Z = \lim_{i \geq 0} Z_i$, it is clear that Z may also include states which are not in fact reachable. Moreover, the approximation errors accumulate over the sequence $Z_0, Z_1, \ldots$. The consequence of this is that the verification problem runs the risk of being answered by a 'false negative': i.e., we may be told that a specification is not satisfied because a violating state is reachable, when, in fact such a state occurs only among those 'extra' states added by the approximation. Wong-Toi [Won95] proposes a solution to this problem in which a succession of over- and under-approximations is computed. If a violating state is reachable in an under-approximation, then the specification is not satisfied. If no violating state is reachable in an over-approximation, then the specification is satisfied. An increasingly accurate sequence of approximations is computed until the verification problem can be answered in this way. However, in some cases, it may be necessary to compute an approximation which captures the set of reachable states exactly, before the verification problem can be answered – this is less efficient than a direct computation of the exact set of reachable states. An alternative approach, which avoids the use of non-convex polyhedra and also avoids the use of approximations, is considered below.

Simulation Graph

In this section we consider a construction, the *simulation graph* [Oli94, DT98], which has appeared often in the literature of dense-time verification under a variety of names, including: *set-graph* [ACD+92, Won95], *zone automaton* [AD96, AK95] and *symbolic semantics* [LPY95, Pet99]. We first give details of the construction and then consider the advantages and disadvantages of its use.

Definition 2.16 (Simulation Graph) Let $\mathcal{A} = (Q, q^{\mathcal{I}}, A, \mathcal{H}, E, I)$ be a TA. Let c be a constant at least as great as $c_{max}(\mathcal{A})$. The *simulation graph* of $\mathcal{A}$ with respect to c, starting at the symbolic state $z_0 = (q_0, \zeta_0)$, is denoted $\mathsf{SG}(\mathcal{A}, c, z_0)$, and is given by $(\mathcal{Z}, z^{\mathcal{I}}, A, \longrightarrow_{\mathsf{sg}})$, where $\mathcal{Z} \subseteq Q \times \mathcal{Z}_{\mathcal{H}}$ and $\longrightarrow_{\mathsf{sg}} \subseteq \mathcal{Z} \times A \times \mathcal{Z}$ are the smallest sets satisfying:

1. $z^{\mathcal{I}} = (q_0, \mathsf{suc}_{\tau}^{q_0}(\zeta_0)) \in \mathcal{Z}$
2. for every $z = (q, \zeta) \in \mathcal{Z}$ and for every $e = (q, \zeta_e, a, \mathsf{H}, q') \in E$, if $\zeta' = \mathsf{close}_c(\mathsf{suc}_{\tau}^{q'}(\mathsf{suc}_e(\zeta))) \neq \emptyset$, then $z' = (q', \zeta') \in \mathcal{Z}$ and $z \xrightarrow{a}_{\mathsf{sg}} z'$ □

Notation. The simulation graph of $\mathcal{A}$ with respect to c, starting at the initial state $(q^{\mathcal{I}}, \mathsf{zero})$, is denoted simply by $\mathsf{SG}(\mathcal{A}, c)$, and $\mathsf{SG}(\mathcal{A})$ denotes $\mathsf{SG}(\mathcal{A}, c_{max}(\mathcal{A}))$.

Intuitively, a simulation graph of $\mathcal{A}$ is constructed by starting with a given symbolic state, and then allowing time to pass – rule 1, above; we then consider all the edges of $\mathcal{A}$ and look for any which can be taken from a node already in the graph; any possible edge transition is taken and time allowed to pass again, the successor node being added to the graph – rule 2, above; c-closure is used to ensure that the graph is finite; this process continues until all possible nodes and edges have been added to the graph.

Let $\mathcal{A} = (Q, q^{\mathcal{I}}, A, \mathcal{H}, E, I)$ be a TA with $\mathcal{T}[\![\mathcal{A}]\!] = (\Sigma, \sigma^{\mathcal{I}}, L, \longrightarrow)$. Let $c \geq c_{max}(\mathcal{A})$ and z a symbolic state. We now state the two key properties of the simulation graph $\mathsf{SG}(\mathcal{A}, c, z) = (\mathcal{Z}, z^{\mathcal{I}}, A, \longrightarrow_{\mathsf{sg}})$.

Proposition 2.5 $\mathsf{SG}(\mathcal{A}, c, z)$ *is finite.*

Proof This follows immediately from the fact that the locations and edges of any TA are finite sets together with proposition 2.2(4). □

Proposition 2.6 (Correctness of simulation graph) *Assume, without loss of generality, that z is the symbolic state (q_0, ζ_0), where ζ_0 denotes the convex $\mathcal{H}$-polyhedron which contains the single point $\mathbf{v}_0$. Then,*

- *(Soundness) whenever $(q_0, \zeta_0) \longrightarrow^*_{\mathsf{sg}} (q_f, \zeta_f)$ then $(q_0, \mathbf{v}_0) \longrightarrow^* (q_f, \mathbf{v}_f)$, for all $\mathbf{v}_f \in \zeta_f$;*
- *(Completeness) whenever $(q_0, \mathbf{v}_0) \longrightarrow^* (q_f, \mathbf{v}_f)$ then $(q_0, \zeta_0) \longrightarrow^*_{\mathsf{sg}} (q_f, \zeta_f)$ for some ζ_f such that $\mathbf{v}_f \in \zeta_f$.*

Proof Straightforward adaptation of theorem 4.1 in Pettersson [Pet99] □

It is clear from Proposition 2.6 that the reachability problem can be solved by searching the simulation graph: in order to determine if $(q', \mathbf{v}')$ is reachable from $(q, \mathbf{v})$ in the transition system of $\mathcal{A}$, it suffices to construct the simulation graph $\mathsf{SG}(\mathcal{A}, c_{max}(\mathcal{A}), z)$ where $z = (q, \{\mathbf{v}\})$; if there is a node (q'', ζ'') such that $q' = q''$ and $\mathbf{v}' \in \zeta''$ then the answer is 'yes', otherwise the answer is 'no'. Figure 2.14 outlines an algorithm which implements this approach.

There are several reasons why reachability analysis based on the simulation graph has been applied successfully:

- Only convex polyhedra are needed in the implementation of the algorithm. We have already seen that there are efficient algorithms for manipulating the DBM representation of convex polyhedra which ensures that the membership test at line 9, the generation of successors at lines 12–13, the test for emptiness at line 13 and the implicit equality test at line 15 can all be computed effectively.

```
1  input
2    𝒜 = (Q, q^𝓘, A, 𝓗, E, I), c = c_max(𝒜),
3     initial state (q, v),  final state (q', v')
4  begin
5    VISITED := {(q, {v})};
6    WAITING := {(q, {v})};
7    while WAITING ≠ ∅ do
8      remove some (q'', ζ'') from WAITING
9      if (q' = q'') ∧ (v' ∈ ζ'')
10       then return  'yes'
11        else
12             succ := {(q_s, ζ_s) | e = (q'', _, _, _, q_s) ∈ E ∧
13                                ζ_s = close_c(suc_τ^{q_s}(suc_e(ζ''))) ≠ ∅};
14             foreach (q_s, ζ_s) ∈ succ do
15               if (q_s, ζ_s) ∉ VISITED
16                 add (q_s, ζ_s)  to VISITED;
17                 add (q_s, ζ_s)  to WAITING
18               fi
19             od
20       fi
21     od;
22     return  'no'
23   end
```

Fig. 2.14: An algorithm for reachability based on the simulation graph

- The reachability test is performed 'on-the-fly', i.e., it is not necessary to generate explicitly the complete product automaton of several TA, nor is it necessary to generate the full state space, before checking whether or not a particular state is reachable. The test (at line 9) can be performed as the state space is constructed, and, indeed, in many cases the algorithm will terminate when only a small fraction of the total number of states has been generated.
- A diagnostic trail can be provided based on the contents of *WAITING*, assuming a stack implementation. In practice, this is of great assistance to the user in the modification of an incorrect system.
- Although the theoretical bound on the size of the simulation graph is exponential in the number of clock regions [ACD$^+$92], in practice, far fewer states are generated than in region graph algorithms – sensitivity to the size of constants in clock constraints is alleviated.

More heuristics

The size of the set *VISITED* of stored states can be reduced by employing two further heuristics, one of which preserves reachability exactly and the other of which preserves it conservatively.

- *Inclusion abstraction* is based on the idea that for two symbolic states z_1 and z_2 such that $z_1 \subseteq z_2$, z_1 need not be explored, since any state in z_1 also belongs to z_2, and any successor of z_1 is also a successor of z_2. The implementation of this idea simply involves a modification of the test at line 15 from $(q_s, \zeta_s) \notin \mathit{VISITED}$ to $\neg\ \exists \zeta \in \mathcal{Z}_{\mathcal{H}}\ .\ (q_s, \zeta) \in \mathit{VISITED} \wedge \zeta_s \subseteq \zeta$. The effect of this is that instead of checking that a successor state is not already in the set of visited states, we check that there is no visited state which 'covers' the successor state, in the sense of having the same control location and being associated with a set of clock valuations which includes all those of the successor. Clearly, this modification may reduce the number of symbolic states which are stored, while ensuring that all, and only, reachable states are considered. This technique is used in the tool UPPAAL [LPY97] and in later versions of KRONOS [BDM+98]. A proof of correctness can be found in [DT98, Tri98].

- *Convex hull abstraction* implements the proposal mentioned above, in the section on avoiding non-convex polyhedra. Once again, the idea is to tolerate an over-approximation of the set of reachable states with the compensation that it is necessary to keep only a single symbolic state (q, ζ) for each control location q. This can be implemented by replacing lines 15–18 with the following:

```
if ∃ζ ∈ Z_H . (q_s, ζ) ∈ VISITED
  then
        if ζ_s ⊈ ζ
          then
                add (q_s, ζ ⊔ ζ_s) to VISITED
                add (q_s, ζ ⊔ ζ_s) to WAITING
        fi
  else
        add (q_s, ζ_s) to VISITED
        add (q_s, ζ_s) to WAITING
fi
```

  The advantages and disadvantages of this approach have been discussed already.

2.7.7 Other attacks on state space explosion

In addition to the symbolic constraint solving algorithms of the previous section, there are several other techniques which have been applied to the problem of state space explosion in the analysis of timed systems. It is outside the scope of this dissertation to give a detailed survey of the literature; instead, we briefly review some of the most significant ideas.

Large grain partitions

As we have seen, the primary objective of any verification algorithm for TA, is to identify a finite partitioning of the infinite space of clock valuations, where the partitioning respects the transition relation. Although the region graph satisfies this requirement, it produces a very fine partitioning with a large number of classes, and so leads to algorithms which often require more computational resources (memory and time) than are available. An interesting question is whether or not it is possible to construct a partitioning with the *smallest number of classes* needed to solve a given verification problem. This question can be answered positively in the case of timed bisimulation equivalence and model checking.

The problem of constructing the quotient of a LTS with respect to an equivalence relation is well-known in the setting of untimed systems, and generic algorithms exist to solve the problem [BFH$^+$92, LY92]. These algorithms have been adapted to TA in [ACD$^+$92, ACH$^+$92], where it is shown how to simultaneously generate and minimise the reachable sub-LTS of a TA. Tripakis and Yovine [TY96] have shown how such minimisation can be performed more efficiently by adapting the idea from [YL93] of avoiding the costly operation of set complementation. Once constructed, the minimal model of a TA may be reduced still further with respect to untimed abstractions, and then checked for equivalence with an untimed specification automaton using a tool such as CADP [FGK$^+$96].

A similar use of large-grained partitions is made by Sokolsky and Smolka [SS95, Sok96] to solve the full model-checking problem for a timed modal μ-calculus. In their approach, partition refinement is applied to a structure which models the 'product' of the symbolic state space and a graph representation of the property specification; their algorithm strives to construct the coarsest possible partitioning which allows the validity of the specification to be decided. Recent work by Lutje-Spelberg et al. [LSTA98] seeks to improve on this approach by using a more compact representation of the set of regions which a partition comprises.

Partial Order Reduction

In asynchronous system models, state space explosion is due partly to the modelling of concurrency by interleaving, whereby the simultaneous occurrence of two or more events is represented by a set of executions which contains all possible orderings of those events. Partial order reduction exploits the observation that it is not always necessary to consider the whole set of such executions, but rather to consider only one representative from each of the classes of 'equivalent' executions [God96, Pel92, Val93]. The application of partial order techniques in tools for the analysis of untimed systems has demonstrated significant state space reduction [Hol96, HP94]. However, similar success has not (yet) been demonstrated for real-time systems. A major difficulty seems to be that the independence of system components is reduced by their need to synchronise with each other in respect of the passage of time [YS96, Pag96, Pag97]. Bengtsson et al. [BJLY98] have recently proposed the use of 'local' clocks in TA, which usually advance independently and are synchronised only when there is a need for communication. Dams et al. [DGKK98] suggest a different approach which incorporates a generalised notion of independence, called 'covering'. Both of these approaches are intended to allow a greater potential for independent behaviour and so to give a coarser partitioning of the set of executions into 'equivalent' classes. So far as we know, there are as yet no successful implementations of partial order reduction methods for dense real-time systems.

Abstraction

All modelling and analysis relies upon abstracting details from the system under investigation, while keeping what is necessary to preserve the properties of interest. An extreme example of abstraction can be seen in approaches which abstract all details of data values from their models, leaving only control information. Less extreme methods of property-preserving abstraction, set within the framework of *abstract interpretation* [CC77], have been proposed in [CGL94, LGS+95, SBLS99]. Application to the verification of LTL properties is discussed in [KP98]. In the case of timed systems, the possibility of abstracting all timing information initially, adding it only when it is known to be needed to demonstrate a given property, has been investigated in [AIKY95]. A different approach is adopted in [TY96], where timed models are constructed initially and then reduced according to a time-abstracting bisimulation. Daws and Tripakis have placed a number of standard techniques for reducing the size of timed systems within the framework of property-preserving abstractions [DT98]. The problem of demonstrating that a timed system model is a correct abstraction of a more concrete system is considered in [TAKB96]. A combination of abstraction with other techniques is the norm. When used in conjunction with modular reasoning and/or theorem proving, it can extend the scope of model checking to systems with infinite state spaces [AAB+99, DF95, RSS95, SS99].

On-the-fly techniques

A system model comprising a set of concurrent tasks exhibits state explosion when the product space is constructed. On-the-fly methods combat state explosion by solving a problem *during* the construction of the product space, rather than *after* it. This means that the full product space may not need to be constructed at all, and so state explosion can be avoided. This technique has been applied successfully in solving reachability problems [JJ91], computing behavioural equivalences and preorders [FM91], checking temporal logic properties [GPVW95, VW86] and minimising state graphs [BFH+92]. Extension of the technique to the solution of similar problems in timed systems has been considered in [BTY97, HKV96, TY96]. On-the-fly methods are most useful when debugging a system, i.e. when checking properties which turn out not to hold. It is difficult to avoid considering all reachable states when checking a true property.

Symbolic methods

The model checking approach was given a big boost by the work of McMillan in the late eighties [McM92]. He discovered that regularly structured state spaces, such as those derived from models of hardware components or communication protocols, can be represented very compactly using binary decision diagrams (BDDs) [Bry86]. The operations needed for model checking can be adapted to work with sets of states, represented as BDDs, rather than individual states. Using this technique, it is possible to verify systems having more than 10^{20} states [BCM$^+$92]. So far, the benefits of such symbolic techniques have not been realised completely in the analysis of timed systems. We consider this problem in more detail in Chapter 5.

Modular/Compositional Verification

We have seen that many systems are implemented and modelled as the composition of several components. Yet another approach to avoiding the construction of the product of the component state spaces is to decompose a global system property into a number of local properties of one or more components, and then to prove that, if the local properties are satisfied, the global property is satisfied also. The intention here is to transform a single, large verification problem into several smaller problems [GL94]. In proving a local property, it is often convenient to *assume* that the environment behaves in a certain manner; it is then necessary for the other system components to *guarantee* this behaviour. The assume/guarantee paradigm is discussed in [HQR98]. The task of decomposing a problem can require significant insight and often defies automation. An approach which can be automated involves the computation of a quotient property with respect to some component which is then removed from the system model, such that proving the quotient property in the reduced model is equivalent to proving the original property in the original model. Iteration of this technique allows a property to verified automatically without having to construct the product state space. This approach has been applied to timed systems [KLL$^+$97] and implemented in the model checker CMC [LL98]. A different approach to automating compositional analysis is introduced in [LAB$^+$98]. In this approach, backwards reachability analysis is performed using only those components which are required to determine the property of interest. Dependency analysis is used to determine which components are relevant. The technique has been applied successfully to embedded systems but its scope has not yet been extended to include timed systems.

Clock reductions

The state explosion problem in timed systems is compounded by the need to take account of clock values [AD94]. The most significant attack on this aspect of the problem is the work of Daws and Yovine [DY96] which shows two methods for reducing the number of clocks needed in a TA:

- *Clock activity reduction* relies on identifying for each TA location those clocks which do not affect the behaviour of the TA before they are reset. Such clocks are said to be *inactive*; the other clocks are said to be *active*. It is only necessary to record the values of the active clocks in each location, so reducing the memory requirements for a set of timed states.

- *Clock equality reduction* is achieved by identifying those clocks whose values are equal in all locations. Such a set of equal-valued clocks can be replaced by a single clock.

Another technique, with a similar purpose, has been introduced in [LLPY97]. The aim here is to replace a DBM M with a minimal set of clock constraints whose solution set is the same as M's. An algorithm is given which computes a minimal set of constraints for any DBM. Memory requirements are reduced by storing this minimal set rather than the full DBM.

2.7.8 Tools

There is now a large number of well-developed computer programs which implement automatic verification of finite state systems (see [CK96] for a survey). Here we concentrate exclusively on those tools which have been shown to be effective in the analysis of dense real-time systems, and which implement the techniques mentioned earlier in this section.

COSPAN has been developed at AT&T and applied to a number of industrial-scale examples, being the basis of the commercial tool FormalCheck. It is based on the theory of ω-automata [Kur94] and allows both enumerative and BDD-based search [TBK95] and homomorphic reductions [TAKB96]. Real-time verification can be performed using either the region graph or the simulation graph [AK95] and timing constraints can be checked incrementally [AIKY95].

HYTECH is a symbolic model checker for linear hybrid automata [HHWT97], which may be seen as generalising TA by allowing the use of continuous variables to model other aspects of system state than time, e.g., temperature or pressure. A system is described as a set of coordinating linear hybrid automata and a symbolic fixpoint computation is used to check the validity of a specification given as an expression in a branching real-time logic which extends TCTL [ACH$^+$95]. The tool has been used to verify a number of small examples [AHP96], including the Philips audio transmission protocol [HWT95]. A key feature of HYTECH is its ability to perform parametric analysis, i.e., to determine the values of design parameters for which a linear hybrid automaton satisfies a temporal logic requirement.

KRONOS was developed originally by Sergio Yovine to implement the model-checking of TA with respect to TCTL specifications using the symbolic method proposed in [HNSY94]. It has since been extended with procedures for: on-the-fly checking of TBA emptiness [BTY97], generation of minimal models by time-abstracting bisimulation [TY96], automatic reduction of the number of clock variables [Daw98b, DY96], inclusion and convex hull abstraction [DT98], and symbolic state space representation using BDDs [BMPY97]. The PhD dissertations of Tripakis [Tri98] and Daws [Daw98a] give detailed descriptions of the most recent technical advances which are implemented in the current version of the tool. The effectiveness of KRONOS has been demonstrated through its application to several case studies, including: the Philips audio transmission protocol [DY95], the CNET protocol [TY98] and the STARI chip [BMPY97].

UPPAAL allows the checking of networks of TA based on reachability analysis of the simulation graph as described earlier. The underlying principles of this approach were described in [YPD94]. The property specification language allows the expression of safety properties, including bounded response, and also simple liveness properties of the form $\exists\Box\, p$ and $\forall\Diamond\, p$, where p is a 'locally' checkable state property. The tool also reports all deadlocked states (i.e., states where no discrete transition will be possible in the future) encountered during a verification. Since its first release in 1995 [BLL^{+}95], UPPAAL has been improved by the introduction of a more efficient representation of clock constraints, a new termination algorithm which requires the storage of fewer visited states [LLPY97], and an improved hash table implementation of the set of visited states [BLL^{+}98]. An important feature of UPPAAL, from the point of view of usability, is a graphical interface which integrates the various features of the tool, such as system description, property specification, simulation and verification. UPPAAL is now sufficiently mature to have been used in a number of industrial case studies, including the analysis of communication protocols such as the Bang & Olufsen audio/video protocol [HSLL97], the Bounded Retransmission protocol [DKRT97], the Dacapo startup protocol [LP97] and a lip synchronisation algorithm for the transmission of multimedia data [BFK^{+}98]. It has also been used in a collaborative project with the automotive industry to assist in the design of a gear controller [LPY98].

Other interesting approaches for which tools exist, although perhaps less well-developed and case-tested than those mentioned above, include: VERITI [Won95] which implements Wong-Toi's method based on successive over- and under-approximation; RT-SPIN [TC96] which extends ProMela, the language of the model-checker SPIN [Hol96], with simple time guards and performs constraint-based reachability analysis on the derived TA; SGM [HW98] which provides an environment in which it is possible to experiment with different combinations of several state graph manipulators [WH98b, WH98a] in order to reduce the size of the state space; PMC [LSTA98] which implements the partition refinement algorithm of Lutje-Spelberg et al.; and CMC [LL98] which implements an improved version of the compositional approach to model checking which was first introduced in [LL95].

2.8 Conclusions

This chapter has reviewed an approach to the formal modelling and analysis of real-time systems. Systems are modelled as labelled timed transition systems over a dense time domain. We have considered the expression of such models using timed process algebra and timed automata. Specifications are given either as expressions in a timed temporal logic such as TCTL, or as specification automata. Analysis techniques are based upon exhaustive state space search, where the major difficulty is the state explosion problem. We have discussed in detail approaches to this problem in which sets of clock valuations are represented as linear constraints, implemented efficiently using DBMs. These languages and methods are the foundation for the work presented in the rest of the dissertation.

This review has necessarily omitted consideration of many other approaches to the modelling and analysis of timed systems, which have appeared in the literature in recent years. We take a small step to fill this gap by briefly mentioning some of them now.

There is a large Petri net community which has established many theoretical results and practical techniques for modelling and analysis. In this context, a variety of timed Petri nets have been suggested for use with timed systems [BD91, Rok93, Sif77].

Graphical modelling languages are of interest since many designers find a visual syntax 'intuitively' clear. Hierarchical structures are needed in order to manage the size of the diagrams for all but the simplest systems. Statecharts [Har87] allow such a hierarchical representation of untimed state transition models. Modecharts [JLM88, JM87, YMW93] extend this approach with explicit timing constraints; another timed Statechart extension, which can be used for modelling hybrid systems also, is given in [KP92].

Lynch and Vaandrager have introduced timed I/O automata, which offer a similar model of timed systems to the timed automata discussed in this chapter; rather than verification via model-checking, they propose refinement and simulation proof techniques [LV95].

Cardell-Oliver [CO92] proposes the use of higher order logic both to model the behaviour of a system and its environment, and also to specify requirements. The task of proving that the combined system and environment satisfy the requirements is supported by the use of a mechanical theorem prover. Hooman [Hoo91, Hoo96] offers a related assertional style of modelling and specification using extended Hoare triples [Hoa69]. Duration Calculus [CHR91, Liu96] is yet another approach in which modelling and verification is conducted within a single logical framework.

Validation of real-time systems by means of formally constructed test suites is considered in [COG98, SVD97].

3. *bCANDLE*: A LOW LEVEL MODELLING LANGUAGE

3.1 Introduction

This chapter introduces a new modelling language called *bCANDLE*. The purpose of *bCANDLE* is to serve as a language for modelling embedded, real-time systems which are organised as a collection of distributed processes communicating via a broadcast network. The broadcast communication primitive adopted by *bCANDLE* is an abstraction of the CAN protocol [ISO92] and *bCANDLE* has been designed specifically with this protocol in mind. It should be possible to adapt the approach described here to the modelling of other styles of broadcast communication but this idea is not pursued in this thesis.

bCANDLE is a system modelling language, i.e., it is a language intended to allow the expression of models of real-time systems. It is not a programming language nor is it a language for specification. It is assumed that programs are developed using a programming language with a range of real-time and communication constructs to simplify the task, and that system requirements are specified more abstractly using some sort of temporal logic language. *bCANDLE* is a low-level language in the sense that it contains a minimal set of constructs for capturing the behaviour of realistic systems. Here *minimal* is not used with some precise meaning, but is intended to imply that it is difficult to see how any of the features of the language could be omitted without adding significantly to the task of the user in constructing models. However, it is possible to imagine higher-level languages which would further ease the task of the model-builder. Such a high-level language is discussed in Chapter 6.

The rest of this chapter is organised as follows: in §3.2 an informal introduction is given to the class of systems to be modelled; the main components of a *bCANDLE* model, namely the data environment, the network model and the process behaviour model are introduced in §3.3, §3.4 and §3.5, respectively; the formal semantics is presented in §3.6 and a simple example of a system model is shown in §3.7. The chapter concludes with a brief discussion of related work in §3.8.

3.2 Informal system model

We address a class of control systems (Figure 3.1) which can be identified by a number of properties:

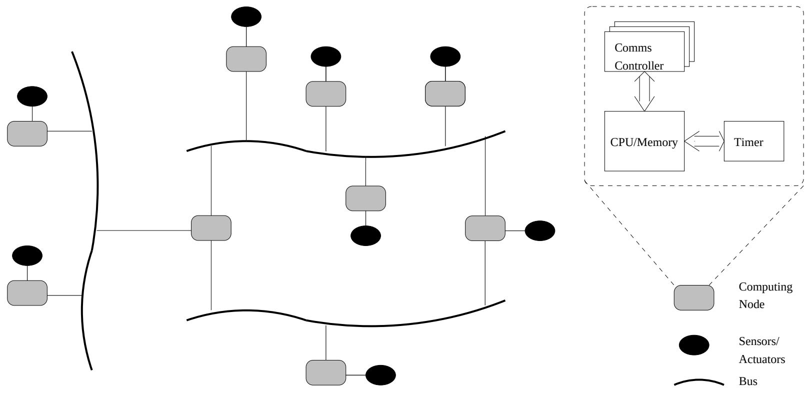


Fig. 3.1: Control system model

- Control is distributed over a set of *processes* which are statically allocated to computing nodes.
- A computing node consists of at least a central processing unit, which has access to some local memory, one or more communication controllers and a programmable timer.
- Several processes may be allocated to a single computing node and share its processing unit using some fixed scheduling policy. The approach taken in this work to the construction of timed models of control systems requires the choice of scheduling policy to be restricted to one which allows static calculation of computation response times: e.g. round-robin or cyclic executive. This allows the effects of scheduling to be accounted for when the model is constructed, without requiring the model to represent the scheduler explicitly. Future work will address how this constraint may be relaxed.
- Processes communicate by using one or more communication channels to send and receive broadcast messages. Each channel implements an abstraction of the CAN protocol, as discussed below.
- Even processes which share a processor communicate by broadcasting messages, rather than by unconstrained access to shared memory, i.e., all processes communicate using (logically) a single mechanism, whether they share a computing node or not. This requirement simplifies the model and can be satisfied with acceptable efficiency in practice. For example, a CAN-style channel can be implemented using shared memory techniques such as condition variables [ISO96] or the mutable variables of Concurrent Haskell [PJF96]. The latency of such a (pseudo-) channel is clearly different from that of a 'real' CAN channel but can be modelled using the same techniques.
- There is no interference between communications on *different* channels, i.e. the transmission of a message on some channel a has no effect on

any other channel b, unless a and b are the same channel. This requirement can be satisfied simply by requiring that every node has a dedicated communication controller for each channel which it uses.

- Each computing node may have access to a number of sensors and actuators which form part of the interface to the controlled system. In the case of multi-tasking, it is assumed that each sensor and actuator is accessed exclusively by a single process.

In constructing a formal model of a system of the sort described above, it is essential to abstract from some of the details, in order to ensure that an analysis of the model is tractable. With this in mind, the following features of an abstract model are identified:

- The *data model* is an abstraction of the set of local memories of the computing nodes. We adopt a single global mapping from data variables to data values and assume that locality is ensured by the syntax of a high-level modelling language.
- The *communication model* abstracts entirely from communication controllers and represents the communication channels only. It is assumed that communication channels operate without errors or failures. Also, the details of bit-level data transmission are abstracted by adopting the assumption that messages are transmitted atomically.
- The *process model* represents the dynamic behaviour of processes, while abstracting from the allocation of processes to computing nodes, and from the scheduling policies adopted by multi-tasking nodes. We assume that an *a priori* analysis accounts for these factors in determining bounds on the completion times of computations. This assumption restricts the systems which can be modelled to those with simple cyclic scheduling policies, but seems essential for tractable analysis.

We present the formal description of each of these aspects of the system models in the following sections.

3.3 The Data Model

Many approaches to the description of concurrent and real-time systems have adopted the point of view that the data environment in which a system acts can be either completely disregarded, or else encoded in the system's behaviour in some way [Mil89, Dav93]. This assumption can simplify the semantic model and its analysis. However, for many systems, the effects of data-dependent behaviour cannot be ignored or abstracted from entirely, and the need to develop an artificial encoding can be tiresome. Therefore we have chosen to include an explicit model of (at least part of) the data environment in our system models, and to employ appropriate abstractions in their analysis when it becomes clearer which properties of the data environment are relevant to the system properties of interest. A similar approach has been adopted in AORTA [BHKR01].

Unlike the LOTOS family of languages [ISO88b, ISO98, Sig98], for example, which give a very detailed description of a particular data sub-language, *bCANDLE* specifies only a minimal set of requirements which a data language must satisfy. In principle, this allows the system modeller to derive models from a variety of different data languages, so long as they are well-defined with respect to the properties described below. For example, a high-level language such as B [Abr96], a programming language such as Spark Ada [Bar96] or a simple guarded command language, as introduced in Chapter 6, can be used as the data language for *bCANDLE*. Although some effort is required to establish the necessary semantic relations, it is rewarded by the flexibility in the choice of language and the simplification in the presentation of *bCANDLE*.

3.3.1 Formal Definition

There are three kinds of syntactic object relating to data which can occur in a *bCANDLE* description: data variables, operation names and predicate names (or guards). The necessary formal definitions are introduced below.

Let Var be a finite set of data variables. Each variable $x \in Var$ takes its value from some non-empty, finite set of values $\mathsf{type}(x) \subseteq V$, where V is the set of data values. We assume that V contains at least the distinguished value $\perp$, where $\perp \notin \bigcup_{x \in Var} \mathsf{type}(x)$, which is taken to be the "undefined" data value. In modelling the behaviour of a system, the current valuation of the data variables is given by a total function from variables to values. The set of valuations is defined by:

$$Valuation \mathrel{\widehat{=}} Var \rightarrow V$$

where for any $\mathsf{val} \in Valuation$ and $x \in Var$, either $\mathsf{val}(x) \in \mathsf{type}(x)$ or $\mathsf{val}(x) = \perp$.

Data operations are modelled as relations on valuations. This allows the use of non-deterministic operation specifications, which are often useful in the construction of abstract system models. Let Ω be a finite set of operation names. Each operation name $\omega \in \Omega$ is interpreted by a total relation on valuations. The set of operations is defined by:

$$Operation \mathrel{\widehat{=}} Valuation \leftrightarrow Valuation$$

where it is required that for every operation o and for every valuation val, there is at least one valuation to which val is related by o, i.e.

$$\forall\, o \in Operation \;.\; \mathrm{dom}(o) = Valuation \;.$$

Predicates on data are modelled simply as the sets of valuations which satisfy them. Let Γ be a finite set of predicate names. Each predicate name $\gamma \in \Gamma$ is interpreted by the set of valuations which satisfy it. The set of predicates is defined:

$$Predicate \mathrel{\widehat{=}} 2^{Valuation}$$

In defining a data environment D with respect to given sets of data variables Var, operation names Ω and predicate names Γ, we say that D is a data environment *over* Var, Ω and Γ, and denote the set of such environments by $DataEnv_{Var,\Omega,\Gamma}$. We can now formally define our notion of a data environment.

Definition 3.1 (Data Environment) Let Var be a finite set of variable names, Ω a finite set of operation names and Γ a finite set of predicate names. Let V be the set of data values. A *data environment* D over Var, Ω and Γ is a tuple

$$D = (\mathsf{type}, \mathsf{operation}, \mathsf{predicate}, \mathsf{val})$$

where $(\mathsf{type}, \mathsf{operation}, \mathsf{predicate}, \mathsf{val}) \in DataEnv_{Var,\Omega,\Gamma}$ iff

- $\mathsf{type} : Var \to 2^V$ is a total function, giving for each variable $x \in Var$, a non-empty, finite set of data values $\mathsf{type}(x)$ ranged over by x;
- $\mathsf{operation} : \Omega \to Operation$ is a total function, giving for each operation name $\omega \in \Omega$, an operation $\mathsf{operation}(\omega)$ which interprets it;
- $\mathsf{predicate} : \Gamma \to Predicate$ is a total function, giving for each predicate name $\gamma \in \Gamma$, a predicate $\mathsf{predicate}(\gamma)$ which interprets it;
- $\mathsf{val} : Var \to V$ is a total function which, for each variable $x \in Var$, gives the current valuation of x, where $\mathsf{val}(x) \in \mathsf{type}(x)$ or $\mathsf{val}(x) = \perp$. $\Box$

We assume that for a given *bCANDLE* description, the interpretations of the variable, operation and predicate names are fixed but that the current valuation may change as the system evolves.

Notation. It is convenient to establish some notational conventions. Let $D = (\mathsf{type}, \mathsf{operation}, \mathsf{predicate}, \mathsf{val})$ be a data environment. Let $x, y \in Var$ be data variables, and $v \in V$ a data value.

- $D.\mathsf{type}$, $D.\mathsf{operation}$, $D.\mathsf{predicate}$ and $D.\mathsf{val}$ denote type, $\mathsf{operation}$, $\mathsf{predicate}$ and val, respectively.
- $D.x$ denotes the value $\mathsf{val}(x)$.
- $D[x := v]$ denotes the data environment $D' = (\mathsf{type}, \mathsf{operation}, \mathsf{predicate}, \mathsf{val}')$ where $\mathsf{val}'(x) = v$ and $\mathsf{val}'(y) = \mathsf{val}(y)$ for all $y \not\equiv x$ ($\equiv$ denotes syntactic identity and $\not\equiv$ its negation).
- $D \xrightarrow{\omega}_{\mathsf{d}} D'$ abbreviates the condition

  $$(\mathsf{val}, \mathsf{val}') \in \mathsf{operation}(\omega) \wedge D' = (\mathsf{type}, \mathsf{operation}, \mathsf{predicate}, \mathsf{val}')$$

  We reserve the operation name ID and require that it is interpreted in any data environment by the operation $\mathsf{operation}(ID)$, where

  $$\mathsf{operation}(ID) \mathrel{\widehat{=}} \{(\mathsf{val}, \mathsf{val}) \mid \mathsf{val} \in Valuation\}$$

  i.e., $\mathsf{operation}(ID)$ is the identity relation on valuations.

- $D \models \gamma$ abbreviates the condition $\mathsf{val} \in \mathsf{predicate}(\gamma)$. We write $D \not\models \gamma$ for $\mathsf{val} \notin \mathsf{predicate}(\gamma)$. We reserve the predicate names *true* and *false* and require that $\forall D . D \models \mathit{true} \wedge D \not\models \mathit{false}$, i.e., *true* and *false* are interpreted in every data environment by $\mathsf{predicate}(\mathit{true})$ and $\mathsf{predicate}(\mathit{false})$, as follows:

$$\begin{array}{rcl} \mathsf{predicate}(\mathit{true}) & \widehat{=} & \mathit{Valuation} \\ \mathsf{predicate}(\mathit{false}) & \widehat{=} & \emptyset \end{array}$$

Let $D_i = (\mathsf{type}_i, \mathsf{operation}_i, \mathsf{predicate}_i, \mathsf{val}_i)$ for $i \in \{1, 2\}$ be two data environments. D_1 and D_2 are said to be *compatible* iff $\mathsf{type}_1 = \mathsf{type}_2$, $\mathsf{operation}_1 = \mathsf{operation}_2$, and $\mathsf{predicate}_1 = \mathsf{predicate}_2$. If D_1 and D_2 are compatible data environments, and additionally $\mathsf{val}_1 = \mathsf{val}_2$, then D_1 and D_2 are said to be *equal*, denoted $D_1 = D_2$. Here it is assumed that all component equalities are defined extensionally in the usual way.

3.4 The Network Model

A network model is an abstraction of a CAN network. It consists of one or more broadcast channels, each implementing an abstraction of the CAN protocol, as follows:

- Each channel operates fault-free, i.e. without the need for error or overload frames.

- A transmitting node only attempts to transmit its highest priority message. (This requirement may seem obvious but in fact needs some effort to satisfy when using some CAN controllers.)

- A node which has messages to transmit attempts to transmit its highest priority message as soon as the channel is free. This implies that each communication controller does not release the channel between transmissions, i.e. it enters a message for arbitration in every arbitration phase if it has a message to transmit. This is important in ensuring that lower priority messages cannot delay the transmission of pending messages of higher priority by beginning transmission during a "gap" between message transmissions.

- It is guaranteed that a message is "simultaneously" accepted either by all nodes which are configured to accept it, or by none of them. There is no possibility of a "partially successful" transmission.

- We assume that we can determine the point during the transmission of a message when a controller begins its acceptance test for the message. In normal operation, a controller which becomes configured to accept messages at any time before it begins its acceptance test, will accept all messages which pass the test thereafter.

In the rest of this section, the structure of channels and networks is considered first; this is followed by a consideration of network behaviour.

3.4.1 Structure

A network is a collection of broadcast channels, each of which is capable of transmitting messages from a single sending node to one or more receiving nodes. Messages comprise a message identifier and a data value. The identifier serves both to identify the type of data contained in the message and also to give a priority to the message for use in the arbitration of transmission collisions. The remainder of this section expands and formalises these ideas.

Messages

The following example will be used throughout this section to illustrate the ideas which are introduced.

Example 3.1 Consider a simple system for monitoring the temperature of a liquid in a chemical tank and the state of a heater which is used to regulate the temperature. The monitoring system receives messages broadcast by a pair of intelligent sensors, one giving the temperature of the liquid and one giving the state of the heater. Let $I = \{HEATER, TEMPERATURE\}$ be the set of message identifiers and $V = \{ON, OFF\} \cup \{-275 \mathinner{..} 275\}$ be the set of data values. A message consisting of message identifier i and data value v is denoted $i.v$. Some possible messages are $HEATER.ON$ and $TEMPERATURE.127$. The set of all possible messages is given by $I \times V$. Notice, however, that some combinations of message identifier and data value are not sensible, e.g., $HEATER.75$ and $TEMPERATURE.OFF$. $\square$

Example 3.1 suggests that it is helpful to identify the messages which a channel is allowed to transmit and leads us to the following definitions.

Definition 3.2 (Messages) Let I be a finite set of message identifiers. Let V be the set of data values. A set of *messages* over I is any finite subset $M \subseteq I \times V$. $\square$

Notation. A *message* $(i, v) \in M$ is written $i.v$.

Message Priority

Referring again to Example 3.1, it is clear that a mechanism is needed to resolve the conflict which arises if the temperature and heater sensors try to transmit their messages simultaneously on the same channel. Such a conflict is resolved, as in the CAN protocol, by assigning a priority ordering to the set of message identifiers associated with the channel. Let $HEATER \prec TEMPERATURE$ denote that $HEATER$ is a higher priority identifier than $TEMPERATURE$. Then, for example, if transmission of the messages $HEATER.ON$ and $TEMPERATURE.127$ is initiated simultaneously, the transmission of the higher priority message $HEATER.ON$ will succeed, and the message

$TEMPERATURE.127$ will compete again for the channel when it next becomes idle.

Definition 3.3 (Priority Ordering) Let I be a set of message identifiers and V a set of data values. Let $M \subseteq I \times V$ be a set of messages. A *priority ordering* is a strict total ordering $\prec : I \leftrightarrow I$ on the message identifiers. The reflexive ordering $\preceq$ is defined as usual: for all $i, i' \in I$,

$$i \preceq i' \Leftrightarrow i \prec i' \vee i = i'.$$

A priority ordering on identifiers induces a partial ordering on the message set M. The derived ordering $\prec : M \leftrightarrow M$ satisfies for all $m, m' \in M$,

$$m \prec m' \Leftrightarrow i \prec i'$$

and the reflexive ordering $\preceq : M \leftrightarrow M$ satisfies

$$m \preceq m' \Leftrightarrow i \preceq i'$$

where $m = i.v$ and $m' = i'.v'$, in each case. These orderings on messages are also referred to as priority orderings, and the overloading is resolved by context. □

Message Transmission

Before transmission of a message can begin, it is required that no other message is already being transmitted on the communication channel; in this case, the channel is said to be *free*. At some time following the commencement of message transmission, all nodes which are listening to the channel perform a test to determine whether or not the message should be *accepted*. This decision depends on the identifier of the transmitted message. If the message identifier matches a message identifier in the acceptance set of a node, then the node accepts the message and the message data is made available to processes residing on it, otherwise the node ignores the message. It is assumed that all nodes perform the acceptance test instantaneously at the same time. At some time after the acceptance test, the channel becomes free again and is available to transmit another message.

Three phases can be clearly identified in the transmission of a message. The *acceptance* phase is the point during the transmission of a message when listening nodes perform their acceptance test. The *pre-acceptance* phase extends from the start of transmission to the point of acceptance. The *post-acceptance* phase extends from the acceptance point to the instant at which the channel next becomes free.

The *transmission latency* of a message is the time which passes during the pre-acceptance and post-acceptance phases of message transmission. It is assumed that upper and lower bounds can be determined for the pre-acceptance and post-acceptance latency of all messages.

	HEATER._ (μs)	*TEMPERATURE*._ (μs)
δ^{lb}	43	55
δ^{ub}	53	65
δ^{lB}	10	10
δ^{uB}	12	12

Tab. 3.1: Example of Transmission Latency Functions

Definition 3.4 (Transmission Latency) Let M be a set of messages. A *transmission latency function for* M is a function $\delta : M \to \mathbb{R}_\infty \times \mathbb{R}_\infty \times \mathbb{R}_\infty \times \mathbb{R}_\infty$, where $\delta(m) = (l, u, l', u')$ implies that $l \leq u$, $l' \leq u'$ and that the lower and upper bounds on the pre- (resp. post-) acceptance phase of the transmission of m are given by l and u (resp. l' and u').

The derived functions $\delta^{\mathsf{lb}}, \delta^{\mathsf{ub}}, \delta^{\mathsf{lB}}, \delta^{\mathsf{uB}} : M \to \mathbb{R}_\infty$ satisfy

$$\begin{aligned} &\forall\, m \in M \;.\; \delta^{\mathsf{lb}}(m) = l \wedge \delta^{\mathsf{ub}}(m) = u \wedge \delta^{\mathsf{lB}}(m) = l' \wedge \delta^{\mathsf{uB}}(m) = u' \\ &\qquad \Leftrightarrow \delta(m) = (l, u, l', u') \end{aligned}$$

□

Notation. The notation lb (resp. $\mathsf{ub}, \mathsf{lB}, \mathsf{uB}$) is used as an abbreviation for $\delta^{\mathsf{lb}}(m)$ (resp. $\delta^{\mathsf{ub}}(m), \delta^{\mathsf{lB}}(m), \delta^{\mathsf{uB}}(m)$) when m is clear from the context.

Example 3.2 Refer again to Example 3.1. Let the transmission latency function be defined as in Table 3.1, where *HEATER*._ stands for the messages *HEATER.ON* and *HEATER.OFF* and *TEMPERATURE*._ stands for any message *TEMPERATURE*.v with $v \in \{-275 \,..\, 275\}$[1]. Some example interpretations are: the lower bound on the time taken to complete the pre-acceptance phase of transmission of the message *HEATER.OFF* is $43\,\mu$s; the upper bound on the time taken to complete the pre-acceptance phase of the transmission of the message *TEMPERATURE*.127 is $65\,\mu$s; and the lower (resp. upper) bound on the time taken to complete the post-acceptance phase of any message is $10\,\mu$s (resp. $12\,\mu$s). □

The *transmission status* of a channel identifies whether the channel is free or is transmitting a message and, if transmitting a message, whether it is in the pre-acceptance, acceptance or post-acceptance phase. If a channel is in its pre-acceptance or post-acceptance phase, the bounds on the time to completion of the phase are deemed to be part of its transmission status, since they determine the time at which the channel may next influence the behaviour of a system. As time passes, the bounds on the time to completion of a phase are reduced equally until the lower bound becomes 0, after which the upper bound may approach the lower bound until it too becomes 0.

[1] Notice that here and throughout, we make use of _ to denote an arbitrary value taken from whatever set of values is appropriate in its context.

Notation	ASCII	Transmission Status
$\downarrow$	`\/`	FREE
$\overset{t_1,t_2}{\rightsquigarrow} m$	`--t1,t2->m`	$(\mathsf{PRE}, m, t_1, t_2)$, pre-acceptance phase of transmission of message m with bounds t_1, t_2 on time to completion, $0 \leq t_1 \leq \mathsf{lb}$, $0 \leq t_2 \leq \mathsf{ub}$
$\uparrow m$	`/\m`	(ACCEPT, m), acceptance point in transmission of m
$m \overset{t_1,t_2}{\rightsquigarrow}$	`m--t1,t2->`	$(\mathsf{POST}, m, t_1, t_2)$, post-acceptance phase of transmission of message m with bounds t_1, t_2 on time to completion, $0 \leq t_1 \leq \mathsf{lB}$, $0 \leq t_2 \leq \mathsf{uB}$

Fig. 3.2: Transmission Status Notation ($m \in M$ and $t_1, t_2 \in \mathbb{R}_\infty$)

Definition 3.5 (Transmission Status) Let M be a set of messages and $\delta : M \rightarrow \mathbb{R}_\infty \times \mathbb{R}_\infty \times \mathbb{R}_\infty \times \mathbb{R}_\infty$ a transmission latency function for M. Let $\{\mathsf{FREE}, \mathsf{PRE}, \mathsf{ACCEPT}, \mathsf{POST}\}$ be a set of distinct constant symbols. The set $Status_{M,\delta}$ is defined:

$$\begin{aligned} Status_{M,\delta} \;\widehat{=}\;& \{\mathsf{FREE}\} \cup PreAcceptance_{M,\delta} \cup (\{\mathsf{ACCEPT}\} \times M) \\ \cup\;& PostAcceptance_{M,\delta} \end{aligned}$$

where, for a message $m \in M$, a lower bound $t_1 \in \mathbb{R}_\infty$ and an upper bound $t_2 \in \mathbb{R}_\infty$:

1. $(\mathsf{PRE}, m, t_1, t_2) \in PreAcceptance_{M,\delta}$ iff $t_1 \leq \delta^{\mathsf{lb}}(m)$, $t_2 \leq \delta^{\mathsf{ub}}(m)$, and $t_2 - t_1 = \delta^{\mathsf{ub}}(m) - \delta^{\mathsf{lb}}(m)$ if $t_1 > 0$, otherwise $t_2 - t_1 \leq \delta^{\mathsf{ub}}(m) - \delta^{\mathsf{lb}}(m)$;

2. $(\mathsf{POST}, m, t_1, t_2) \in PostAcceptance_{M,\delta}$ iff $t_1 \leq \delta^{\mathsf{lB}}(m)$, $t_2 \leq \delta^{\mathsf{uB}}(m)$, and $t_2 - t_1 = \delta^{\mathsf{uB}}(m) - \delta^{\mathsf{lB}}(m)$ if $t_1 > 0$, otherwise $t_2 - t_1 \leq \delta^{\mathsf{uB}}(m) - \delta^{\mathsf{lB}}(m)$. $\square$

Notation. In the rest of the dissertation, the notation shown in Figure 3.2 is often used as a shorter and more suggestive notation for transmission status.

Message Queues

If it is attempted to transmit a message on a channel which is not free, the message must be stored and offered for transmission again some time after the current transmission has finished. Since messages succeed in their transmission attempts according to their priority, the storing of messages is modelled naturally as a priority ordered queue. If an attempt is made to transmit a message m, whose identifier is the same as that of another message m' which is already in the message queue, then m replaces m' in the queue and m' is lost forever, i.e. m' is 'overwritten' by m. This represents the behaviour of most implementations of the CAN protocol.

Definition 3.6 (Message Queue) Let I be a finite set of message identifiers and V a set of data values. Let $M \subseteq I \times V$ be a set of messages and $\prec$ a priority ordering for M. $Queue_{M,\prec}$ is defined to be the set of all sequences over the message set M which satisfy the following two invariant properties:

$$\forall\, u \in Queue_{M,\prec};\ j, j' \in \operatorname{dom} u \;.\; j < j' \Rightarrow u(j) \prec u(j') \tag{3.1}$$

$$\forall\, u \in Queue_{M,\prec};\ i \in I \;.\; \#\{j \mid j \in \operatorname{dom} u \wedge u(j) = i.\_\} \leq 1 \tag{3.2}$$

i.e. All message queues preserve the priority ordering of messages and contain at most one message with a given message identifier. □

A corollary of property 3.2 is that all message queues are of finite length.

Proposition 3.1 *Let I be a finite set of message identifiers and V a set of data values. Let $M \subseteq I \times V$ be a set of messages and $\prec$ a priority ordering for M. For all $u \in Queue_{M,\prec}$, u is of finite length.*

Proof Immediate from property 3.2 of Definition 3.6 and the fact that I is finite. □

Notation. An empty queue is denoted $\langle\rangle$. A queue with highest priority message m and remaining messages u is written $m{:}u$.

The queueing of a message is modelled by the following operation.

Definition 3.7 (Message Queue Insertion) Let I be a set of message identifiers and V a set of data values. Let $M \subseteq I \times V$ be a set of messages and $\prec$ a priority ordering for M. The insertion operator $\hookleftarrow$: $Queue_{M,\prec} \times M \rightarrow Queue_{M,\prec}$ is defined:

$$u \hookleftarrow i.v = \begin{cases} \langle i.v \rangle & \text{, if } u = \langle\rangle \\ i.v{:}u' & \text{, if } u = i.\_{:}u' \\ i.v{:}m{:}u' & \text{, if } u = m{:}u' \wedge i.v \prec m \\ m{:}(u' \hookleftarrow i.v) & \text{, if } u = m{:}u' \wedge m \prec i.v \end{cases}$$

□

It is easy to show that $\hookleftarrow$ preserves the message queue invariants.

Proposition 3.2 *Let M be a set of messages and $\prec$ a priority ordering for M. For all $u \in Queue_{M,\prec}$ and all $m \in M$, $u \hookleftarrow m$ satisfies properties 3.1 and 3.2 of Definition 3.6.*

Proof Induction on the length of u. □

Channels

All of the prerequisites for the definition of communication channels have now been introduced.

Definition 3.8 (Channel) Let I be a set of message identifiers. Let V be the set of data values. A *channel* over I is a tuple $(M, \prec, \delta, s, u)$. The set of channels over I is denoted $\mathit{Channel}_I$, and $(M, \prec, \delta, s, u) \in \mathit{Channel}_I$ iff

- $M \subseteq I \times V$ is a set of messages,
- $\prec: I \leftrightarrow I$ is a priority ordering,
- $\delta : M \to \mathbb{R}_\infty \times \mathbb{R}_\infty \times \mathbb{R}_\infty \times \mathbb{R}_\infty$ is a transmission latency function,
- $s : \mathit{Status}_{M,\delta}$ is a transmission status
- $u : \mathit{Queue}_{M,\prec}$ is a message queue □

Let $(M, \prec, \delta, s, u)$ be a channel. It is assumed that M, $\prec$ and δ are *static*, i.e., defined at system initialisation and unchanging thereafter. On the other hand, s and u are used to model the current transmission status and message queue of a channel as a system evolves, and are therefore *dynamic*.

The variables η, η', η_1 etc. are used to range over channels. Let $\eta_i = (M_i, \prec_i, \delta_i, s_i, u_i)$ be two channels. η_1 and η_2 are said to be *equal*, denoted $\eta_1 = \eta_2$, iff $M_1 = M_2$, $\prec_1 = \prec_2$, $\delta_1 = \delta_2$, $s_1 = s_2$ and $u_1 = u_2$, where the component equalities are defined extensionally as usual.

Networks

A network is a collection of channels in which each channel is associated with its own unique identifier.

Definition 3.9 (Network) Let K be a finite set of channel identifiers and I a finite set of message identifiers. A *network* N over K and I is a mapping $N : K \to \mathit{Channel}_I$. The set of networks over K and I is denoted $\mathit{Network}_{K,I}$, where $\mathit{Network}_{K,I} \mathrel{\widehat{=}} K \to \mathit{Channel}_I$. □

Notation. Let K be a set of channel identifiers and N a network over K. Let $k \in K$ be a channel identifier. We write N_k for the function application $N(k)$, i.e. N_k denotes the channel associated with the identifier k in the network N.

Network equality is defined extensionally as usual.

3.4.2 Behaviour

Each channel in a network can act independently by making a discrete change in its transmission status or its message queue. Alternatively, the state of the whole network may be affected as time progresses. We consider first the modelling of discrete state changes.

When a channel η makes a discrete change, it gives rise to a new network state in which η is in its new state and the state of all the other channels is the same as before. It is convenient to introduce an operator which models the effect on a network of a change of state in a single channel.

Definition 3.10 (Network Update) Let K be a set of channel identifiers and I a set of message identifiers. Let $N \in \mathit{Network}_{K,I}$ be a network. Let $\eta \in \mathit{Channel}_I$ be a channel. The notation $N[k := \eta]$ denotes the network N', where $N'_k = \eta$ and $N'_{k'} = N_{k'}$, for all $k' \in K \setminus \{k\}$. □

Network behaviour is modelled by a relation $\longrightarrow_{\mathsf{n}} \subseteq \mathit{Network} \times (A_{\mathsf{n}} \cup \mathbb{R}) \times \mathit{Network}$ which represents possible changes in network state. As usual, $(N, \lambda_{\mathsf{nt}}, N') \in \longrightarrow_{\mathsf{n}}$ is written $N \xrightarrow{\lambda_{\mathsf{nt}}}_{\mathsf{n}} N'$ and represents a change of state from N to N' annotated with the label λ_{nt} which ranges over $A_{\mathsf{n}} \cup \mathbb{R}$. A_{n} is the set of network action labels which are used to annotate discrete state changes. Elements of $\mathbb{R}$ are used to annotate state changes due to the passage of time.

Definition 3.11 (Network Action Labels) Let V be the set of data values. The set A_{n} of *network action labels* over K and I is defined by:

$$\begin{array}{rll} A_{\mathsf{n}} & \widehat{=} \; \{k \rightsquigarrow i.v \mid k \in K \wedge i \in I \wedge v \in V\} & \text{(* pre-acceptance *)} \\ & \cup \; \{k \uparrow i.v \mid k \in K \wedge i \in I \wedge v \in V\} & \text{(* acceptance *)} \\ & \cup \; \{i.v \rightsquigarrow k \mid k \in K \wedge i \in I \wedge v \in V\} & \text{(* post-acceptance *)} \\ & \cup \; \{k \downarrow \mid k \in K\} & \text{(* free *)} \end{array}$$

where K and I are sets of channel identifiers and message identifiers, respectively. □

Notation. In describing the behaviour of a network, it is often convenient to mention only the dynamic components of each network channel. For example, the channel $(M, \prec, \delta, s, u)$ may be written (s, u). The static components ($M, \prec$ and δ) are inferred from the context.

The relation $\longrightarrow_{\mathsf{n}}$ is given by a set of Plotkin-style inference rules, as introduced below.

Pre-Acceptance

A channel which has a non-empty queue of pending messages, and whose transmission status is free, starts transmission of its highest priority message. The transmitted message m is removed from the pending queue and the transmission status of the channel shows that it is in the pre-acceptance phase of transmission of m. The lower and upper bounds on the time to completion of the pre-acceptance phase are given by $\delta^{\mathsf{lb}}(m)$ and $\delta^{\mathsf{ub}}(m)$, respectively. This is expressed formally by the rule **N.1** below:

$$\textbf{N.1}\ \frac{N_k = (\Downarrow, m : u)}{N \xrightarrow{k \rightsquigarrow m}_{\mathsf{n}} N[k := (\overset{\mathsf{lb},\mathsf{ub}}{\rightsquigarrow} m, u)]}$$

Acceptance

When the lower bound on the time to completion of the pre-acceptance phase of the transmission of the message m becomes 0, a channel can change state to the acceptance phase of the transmission of m. This is expressed formally by the rule **N.2** below:

$$\textbf{N.2}\ \frac{N_k = (\overset{0,-}{\rightsquigarrow} m, u)}{N \xrightarrow{k \Uparrow m}_{\mathsf{n}} N[k := (\Uparrow m, u)]}$$

Post-Acceptance

A channel in the acceptance phase of the transmission of a message m can enter the post-acceptance phase, whose bounds on time to completion are given by $\delta^{\mathsf{lB}}(m)$ and $\delta^{\mathsf{uB}}(m)$. This is expressed formally by the rule **N.3** below:

$$\textbf{N.3}\ \frac{N_k = (\Uparrow m, u)}{N \xrightarrow{m \rightsquigarrow k}_{\mathsf{n}} N[k := (m \overset{\mathsf{lB},\mathsf{uB}}{\rightsquigarrow}, u)]}$$

Free

When the lower bound on the time to completion of the post-acceptance phase of the transmission of the message m becomes 0, a channel can change its state to free. This is expressed formally by the rule **N.4** below:

$$\textbf{N.4}\ \frac{N_k = (\_ \overset{0,-}{\rightsquigarrow}, u)}{N \xrightarrow{k \Downarrow}_{\mathsf{n}} N[k := (\Downarrow, u)]}$$

Time Progress

In order for time progress to be possible for a network N, it must be possible for *all* channels in N; the rate of progress is the same in all channels. A free channel allows time to pass indefinitely if its message queue is empty, but must begin transmission of its highest priority message without delay otherwise. Similarly, a channel at its acceptance point does not allow time to pass. Passage of time in the pre-acceptance and post-acceptance phases of message transmission is bounded by the time to completion of the phase. We define a function tcp which determines, for any given channel, the maximum amount of time that can pass before the channel must make a discrete state change.

Definition 3.12 (Time Progress) Let η be a channel. The maximum time progress allowed for η is given by $\mathsf{tcp}(\eta)$, where

$$\mathsf{tcp}(\downharpoonleft, \langle\rangle) \mathrel{\widehat{=}} \infty \quad \mathsf{tcp}(\downharpoonleft, \_ : \_) \mathrel{\widehat{=}} 0 \quad \mathsf{tcp}(\overset{\_,t}{\rightsquigarrow} \_, \_) \mathrel{\widehat{=}} t$$
$$\mathsf{tcp}(\uparrow \_, \_) \mathrel{\widehat{=}} 0 \quad \mathsf{tcp}(\_ \overset{\_,t}{\rightsquigarrow}, \_) \mathrel{\widehat{=}} t$$

Let K be a set of channel identifiers and N a network over K. The maximum time progress allowed for N is given by $\mathsf{tcp}(N)$, which is defined by

$$\mathsf{tcp}(N) \mathrel{\widehat{=}} \mathsf{min}\{\mathsf{tcp}(N_k) \mid k \in K\}$$

where $\mathsf{min}\, S$ returns the minimum of the finite, ordered set S. $\square$

When time passes, the state of the network changes accordingly. We use the notation $\eta + t$ (resp. $N + t$) to denote the state of the channel η (resp. network N) after the passage of t units of time.

Definition 3.13 (Effect of time progress: Channels) Let η be a channel. Let $t \in \mathbb{R}$. The state of the channel η after the progress of t units of time is denoted $\eta + t$, where $\eta + t$ is defined by:

$$\begin{array}{rcl}
(\downharpoonleft, \langle\rangle) + t & \widehat{=} & (\downharpoonleft, \langle\rangle) \\
(\downharpoonleft, m : u) + t & \widehat{=} & \mathsf{if}\ t = 0\ \mathsf{then}\ (\downharpoonleft, m : u)\ \mathsf{else}\ \bot \\
(\overset{t_1,t_2}{\rightsquigarrow} m, u) + t & \widehat{=} & \mathsf{if}\ t \leq t_2\ \mathsf{then}\ (\overset{t_1 \dot{-} t, t_2 \dot{-} t}{\rightsquigarrow} m, u)\ \mathsf{else}\ \bot \\
(\uparrow m, u) + t & \widehat{=} & \mathsf{if}\ t = 0\ \mathsf{then}\ (\uparrow m, u)\ \mathsf{else}\ \bot \\
(m \overset{t_1,t_2}{\rightsquigarrow}, u) + t & \widehat{=} & \mathsf{if}\ t \leq t_2\ \mathsf{then}\ (m \overset{t_1 \dot{-} t, t_2 \dot{-} t}{\rightsquigarrow}, u)\ \mathsf{else}\ \bot
\end{array}$$

where $\eta + t = \bot$ is interpreted to mean that the result is not a well-defined channel. $\square$

Proposition 3.3 *Let η be a channel. Then, $\eta + 0 = \eta$.*

Proof Immediate from Definition 3.13. $\square$

Definition 3.14 (Effect of time progress: Networks) Let K be a set of channel identifiers and N a network over K. Let $t \in \mathbb{R}$. The state of the network N after the progress of t units of time is denoted by $N + t$, where

$$N + t \quad \widehat{=} \quad \begin{cases} \{k \mapsto (N_k + t) \mid k \in K\}, & \forall\, k \in K \;.\; N_k + t \neq \bot \\ \bot, & \text{otherwise} \end{cases}$$

where $N + t = \bot$ is interpreted to mean that the result is not a well-defined network. $\square$

Proposition 3.4 *Let N be a network. Then, $N + 0 = N$.*

Proof By Definition 3.14 and Proposition 3.3. $\Box$

Proposition 3.5 *For any network N and time $t \in \mathbb{R}$, if $0 \leq t \leq \mathsf{tcp}(N)$ then $N + t$ is well-defined, i.e., $N + t \neq \bot$.*

Proof Immediate from definitions 3.12 and 3.14. $\Box$

If all channels in a network N can allow time to progress by t time units, then N can allow time to progress by t time units, changing state to become $N + t$. This is expressed formally by the rule **N.5** below:

$$\textbf{N.5}\ \frac{0 \leq t \leq \mathsf{tcp}(N)}{N \xrightarrow{t}_{\mathsf{n}} N + t}$$

Proposition 3.6 *Let N be a network. Then, $N \xrightarrow{0}_{\mathsf{n}} N$.*

Proof Immediate from N.5, Proposition 3.4 and Definition 3.12. $\Box$

Summary

For ease of reference, we summarise the discussion of network behaviour by giving the following definition:

Definition 3.15 (Network Behaviour) Let V be the set of data values. Let K and I be sets of channel identifiers and message identifiers, respectively. Let A_{n} be the set of action labels over K and I. The *network behaviour relation* $\longrightarrow_{\mathsf{n}} \subseteq \mathit{Network}_{K,I} \times (A_{\mathsf{n}} \cup \mathbb{R}) \times \mathit{Network}_{K,I}$ is given by the rules of Figure 3.3, where for all $N, N' \in \mathit{Network}$ and $\lambda_{\mathsf{nt}} \in A_{\mathsf{n}} \cup \mathbb{R}$, $N \xrightarrow{\lambda_{\mathsf{nt}}}_{\mathsf{n}} N'$ iff this can be inferred from the rules **N.1** – **N.5**. $\Box$

It is clear that $\longrightarrow_{\mathsf{n}}$ is well-behaved in that, if $N \xrightarrow{\lambda_{\mathsf{nt}}}_{\mathsf{n}} N'$, then N' is a well-defined network and all of the static components are the same in N and N'.

Proposition 3.7 *Let K be a set of channel identifiers and N a network over K. If $N \xrightarrow{\lambda_{\mathsf{nt}}}_{\mathsf{n}} N'$, then $N' \neq \bot$, and, for any channel identifier k, where $N_k = (M, \prec, \delta, s, u)$ and $N'_k = (M', \prec', \delta', s', u')$, the following properties hold:*

1. *$M = M'$, $\prec\ =\ \prec'$ and $\delta = \delta'$;*
2. *$s' \in \mathit{Status}_{M,\delta}$*
3. *$u' \in \mathit{Queue}_{M,\prec}$*

$$\textbf{N.1}\ \frac{N_k = (\Downarrow, m : u)}{N \xrightarrow{k \rightsquigarrow m}_{\mathsf{n}} N[k := (\overset{\mathsf{lb,ub}}{\rightsquigarrow} m, u)]}$$

$$\textbf{N.2}\ \frac{N_k = (\overset{0,-}{\rightsquigarrow} m, u)}{N \xrightarrow{k \uparrow m}_{\mathsf{n}} N[k := (\uparrow m, u)]}$$

$$\textbf{N.3}\ \frac{N_k = (\uparrow m, u)}{N \xrightarrow{m \rightsquigarrow k}_{\mathsf{n}} N[k := (m \overset{\mathsf{lB,uB}}{\rightsquigarrow}, u)]}$$

$$\textbf{N.4}\ \frac{N_k = (\_ \overset{0,-}{\rightsquigarrow}, u)}{N \xrightarrow{k \Downarrow}_{\mathsf{n}} N[k := (\Downarrow, u)]}$$

$$\textbf{N.5}\ \frac{0 \leq t \leq \mathsf{tcp}(N)}{N \xrightarrow{t}_{\mathsf{n}} N + t}$$

Fig. 3.3: Rules for Network Behaviour

Proof The proofs of all properties follow directly from case analysis of the rules **N.1** – **N.5** by which $N \xrightarrow{\lambda_{\mathsf{nt}}}_{\mathsf{n}} N'$ is inferred. That $N' \neq \perp$ is immediate from **N.5** and Proposition 3.5. Properties 1–3 follow from **N.1** – **N.5** and Definitions 3.5 and 3.6. □

Example of network behaviour

We now give an example of the possible behaviour of a simple network.

Example 3.3 Consider a network consisting of a single channel which can transmit messages of type *temperature* or of type *pressure*. Assume that the values transmitted are abstractions of actual sensor readings, where 0 represents a reading in the low range, 1 a reading in the normal range, and 2 a reading in the high range. The network can be defined as follows.

The message identifiers are given by the set $I = \{temperature, pressure\}$, with priority order $\prec$ given by $temperature \prec pressure$. The set of data values is $V = \{0, 1, 2\}$ and the set of messages is $M = I \times V$. There is a single channel identifier given by the set $K = \{k\}$. The function δ specifies transmission latencies in μsecs, as follows:

	temperature._	*pressure*._
δ^{lb}	43	32
δ^{ub}	53	42
δ^{lB}	10	10
δ^{uB}	12	12

The network is $N = \{k \mapsto (M, \prec, \delta, \Downarrow, \langle temperature.1, pressure.0\rangle)\}$. Notice that we are assuming that the messages *temperature*.1 and *pressure*.0 have

$$
\begin{array}{lll}
(\downharpoonleft, \langle \mathit{temperature}.1, \mathit{pressure}.0\rangle) & \xrightarrow{k \rightsquigarrow \mathit{temperature}.1}_{\mathsf{n}} & (N.1) \\
(\overset{43,53}{\rightsquigarrow} \mathit{temperature}.1, \langle \mathit{pressure}.0\rangle) & \xrightarrow{47}_{\mathsf{n}} & (N.5) \\
(\overset{0,6}{\rightsquigarrow} \mathit{temperature}.1, \langle \mathit{pressure}.0\rangle) & \xrightarrow{k \upharpoonright \mathit{temperature}.1}_{\mathsf{n}} & (N.2) \\
(\upharpoonright \mathit{temperature}.1, \langle \mathit{pressure}.0\rangle) & \xrightarrow{\mathit{temperature}.1 \rightsquigarrow k}_{\mathsf{n}} & (N.3) \\
(\mathit{temperature}.1 \overset{10,12}{\rightsquigarrow}, \langle \mathit{pressure}.0\rangle) & \xrightarrow{12}_{\mathsf{n}} & (N.5) \\
(\mathit{temperature}.1 \overset{0,0}{\rightsquigarrow}, \langle \mathit{pressure}.0\rangle) & \xrightarrow{k\downharpoonleft}_{\mathsf{n}} & (N.4) \\
(\downharpoonleft, \langle \mathit{pressure}.0\rangle) & \xrightarrow{k \rightsquigarrow \mathit{pressure}.0}_{\mathsf{n}} & (N.1) \\
(\overset{32,42}{\rightsquigarrow} \mathit{pressure}.0, \langle\rangle) & \xrightarrow{32}_{\mathsf{n}} & (N.5) \\
(\overset{0,10}{\rightsquigarrow} \mathit{pressure}.0, \langle\rangle) & \xrightarrow{k \upharpoonright \mathit{pressure}.0}_{\mathsf{n}} & (N.2) \\
(\upharpoonright \mathit{pressure}.0, \langle\rangle) & \xrightarrow{\mathit{pressure}.0 \rightsquigarrow k}_{\mathsf{n}} & (N.3) \\
(\mathit{pressure}.0 \overset{10,12}{\rightsquigarrow}, \langle\rangle) & \xrightarrow{11}_{\mathsf{n}} & (N.5) \\
(\mathit{pressure}.0 \overset{0,1}{\rightsquigarrow}, \langle\rangle) & \xrightarrow{k\downharpoonleft}_{\mathsf{n}} & (N.4) \\
(\downharpoonleft, \langle\rangle) & \xrightarrow{5}_{\mathsf{n}} & (N.5) \\
(\downharpoonleft, \langle\rangle) & \xrightarrow{2}_{\mathsf{n}} & (N.5) \\
(\downharpoonleft, \langle\rangle) & \xrightarrow{500}_{\mathsf{n}} & (N.5) \\
\vdots & &
\end{array}
$$

Fig. 3.4: Example of network behaviour

already been placed in the message queue of k.

Now a possible trace of the network behaviour from this initial state is given in Figure 3.4. □

The behaviour given in this example is very simple, since messages are transmitted but not received. How a network interacts with receiving processes is considered in the following section.

3.5 The Process Model

We use a simple process language to describe the behaviour of processes. In choosing the operators of the language, we have been concerned to identify a small set which allows us to express naturally the behavioural models in which we are interested, while allowing the definition of a timed transition semantics in a direct manner. The syntax and informal semantics of the language are presented in this section. Section 3.6 gives a formal semantics and the work of the numerous researchers which has influenced the language design is discussed in §3.8.

3.5.1 Syntax

Definition 3.16 (Process terms) Let K and I be finite sets of channel identifiers and message identifiers, respectively. Let Var be a finite set of data variables, Ω a finite set of operation names and Γ a finite set of predicate names. Finally, let $\mathbb{R}_\infty$ be the time domain and $\mathcal{X}$ a countable set of process variables. The set of *process terms* over K, I, Var, Ω and Γ is denoted $Proc^+_{K,I,Var,\Omega,\Gamma}$, and is defined inductively by:

$$
\begin{array}{lcll}
P & ::= & k!i.x & \text{(* send broadcast message *)} \\
 & | & k?i.x & \text{(* receive broadcast message *)} \\
 & | & [\omega : t_1, t_2] & \text{(* time-bounded computation *)} \\
 & | & \gamma \to P & \text{(* data guard *)} \\
 & | & P \,;\, P & \text{(* sequential composition *)} \\
 & | & P + P & \text{(* non-deterministic choice *)} \\
 & | & P \,[>\, P & \text{(* interrupt *)} \\
 & | & P \mid P & \text{(* parallel composition *)} \\
 & | & \mathrm{rec}\, X.P & \text{(* recursion *)} \\
 & | & X & \text{(* process variable *)}
\end{array}
$$

where $k \in K$, $i \in I$, $x \in Var$, $\omega \in \Omega$, $\gamma \in \Gamma$, $X \in \mathcal{X}$ and $t_1 \leq t_2 \in \mathbb{R}_\infty$. □

Notation. The subscript is dropped from $Proc^+_{K,I,Var,\Omega,\Gamma}$ if it is not relevant or can be inferred from the context.

Terms of the form $k!i.x$, $k?i.x$ and $[\omega : t_1, t_2]$ are called *basic terms*. We use variables $\beta, \beta_1, \beta_2 \ldots$ to range over basic terms. The precedence of the operators, from high to low is: $\to, ;, +, [>, \mathrm{rec}\,, |$. We use a number of syntactic abbreviations:

$$
\begin{array}{ccc}
[\omega : t_1] \,\widehat{=}\, [\omega : t_1, t_1] & [t_1] \,\widehat{=}\, [ID : t_1] & [t_1, t_2] \,\widehat{=}\, [ID : t_1, t_2] \\
\mathrm{idle} \,\widehat{=}\, [ID : \infty] & \mathrm{null} \,\widehat{=}\, [ID : 0] &
\end{array}
$$

Free and *bound* variables are defined as usual.

Definition 3.17 (Free and bound variables)
Let $P \in Proc^+$ and $\bowtie \in \{;, +, [>, |\}$. The *free (process) variables* and *bound (process) variables* of P are given by $\mathsf{fv}(P)$ and $\mathsf{bv}(P)$ respectively, which are defined as the least sets satisfying:

$$
\begin{array}{rr}
\mathsf{fv}(k!i.x) = \emptyset & \mathsf{bv}(k!i.x) = \emptyset \\
\mathsf{fv}(k?i.x) = \emptyset & \mathsf{bv}(k?i.x) = \emptyset \\
\mathsf{fv}([\omega : t_1, t_2]) = \emptyset & \mathsf{bv}([\omega : t_1, t_2]) = \emptyset \\
\mathsf{fv}(\gamma \to P) = \mathsf{fv}(P) & \mathsf{bv}(\gamma \to P) = \mathsf{bv}(P) \\
\mathsf{fv}(X) = \{X\} & \mathsf{bv}(X) = \emptyset \\
\mathsf{fv}(\mathrm{rec}\, X.P) = \mathsf{fv}(P) \setminus \{X\} & \mathsf{bv}(\mathrm{rec}\, X.P) = \mathsf{bv}(P) \cup \{X\} \\
\mathsf{fv}(P \bowtie Q) = \mathsf{fv}(P) \cup \mathsf{fv}(Q) & \mathsf{bv}(P \bowtie Q) = \mathsf{bv}(P) \cup \mathsf{bv}(Q)
\end{array}
$$

□

Definition 3.18 (Closed term) For any $P \in \mathit{Proc}^+$, P is a *closed* term if $\mathsf{fv}(P) = \emptyset$. $\square$

The use of sequential composition as a basic operator, rather than action prefix, requires some care in the definition of guarded terms.

Definition 3.19 (Guarding, Guarded process variable, Guarded term) Any basic term $\beta \in \mathit{Proc}^+$ is *guarding.* A term of the form $P_1 \,;\, P_2$ or $P_1 \mid P_2$ is *guarding* if P_1 is guarding or P_2 is guarding. A term of the form $P_1 + P_2$ or $P_1 \,[\!>\, P_2$ is *guarding* if P_1 and P_2 are guarding. A term of the form $\mathrm{rec}\, X.P$ is *guarding* if P is guarding.

Let $P \in \mathit{Proc}^+$ be a term containing one or more *occurrences* of a variable $X \in \mathcal{X}$. An occurrence of X is *guarded* in P if P has a subterm of the form $P_1 \,;\, P_2$ where the occurrence of X is contained in P_2 and P_1 is guarding. Otherwise this occurrence of X is *unguarded* in P. A process variable X is *guarded* in a term P if every occurrence of X is guarded in P. A term P is *guarded* if all of its process variables are guarded in P. $\square$

Definition 3.20 (Closed, guarded terms) The set $\mathit{Proc} \subset \mathit{Proc}^+$ is defined to be the set of closed, guarded terms in Proc^+. $\square$

Equational Presentation

In practice, the use of the recursion operator $\mathrm{rec}\, X.P$ is often inconvenient and the use of a set of mutually recursive equations is preferable. We will use whatever form is more convenient in its context and regard a term defined using a set of simultaneous equations as denoting its corresponding term given in terms of the recursion operator.

Definition 3.21 Let E be a finite set of equations $\{X_1 \mathrel{\widehat{=}} P_1, X_2 \mathrel{\widehat{=}} P_2, \ldots, X_n \mathrel{\widehat{=}} P_n\}$ where $\bigcup_{i\in\{1..n\}} \mathsf{fv}(P_i) \subseteq \{X_1, \ldots, X_n\}$. Let P be a process term, $\mathsf{fv}(P) \subseteq \{X_1, .., X_n\}$. Then, the process term corresponding to P is given by the normal form of P^E under the rewrite relation $\longrightarrow_{\mathsf{rw}}$, defined by:

$$X^E \longrightarrow_{\mathsf{rw}} \begin{cases} \mathrm{rec}\, X.(P^{E\setminus\{X \mathrel{\widehat{=}} P\}}) & \text{if } (X \mathrel{\widehat{=}} P) \in E, \\ X & \text{otherwise} \end{cases}$$

$$\begin{array}{llllll}
(k!i.x)^E & \longrightarrow_{\mathsf{rw}} & k!i.x & (k?i.x)^E & \longrightarrow_{\mathsf{rw}} & k?i.x \\
([\omega : t_1, t_2])^E & \longrightarrow_{\mathsf{rw}} & [\omega : t_1, t_2] & (\gamma \rightarrow P)^E & \longrightarrow_{\mathsf{rw}} & \gamma \rightarrow (P)^E \\
(P \,;\, Q)^E & \longrightarrow_{\mathsf{rw}} & (P)^E \,;\, (Q)^E & (P + Q)^E & \longrightarrow_{\mathsf{rw}} & (P)^E + (Q)^E \\
(P \,[\!>\, Q)^E & \longrightarrow_{\mathsf{rw}} & (P)^E \,[\!>\, (Q)^E & (P \mid Q)^E & \longrightarrow_{\mathsf{rw}} & (P)^E \mid (Q)^E
\end{array}$$

$\square$

Example 3.4 Consider the equational presentation

```
A
where
A = [a:1] ; B
B = [b:1] ; A
```

Let $E_A \mathrel{\widehat{=}} \{A \mathrel{\widehat{=}} [a:1] \,;\, B\}$, $E_B \mathrel{\widehat{=}} \{B \mathrel{\widehat{=}} [b:1] \,;\, A\}$, and $E \mathrel{\widehat{=}} E_A \cup E_B$.

Then, with respect to E, A is taken to stand for the term

$$\begin{array}{lll}
A^E & \longrightarrow_{\mathsf{rw}} & \mathrm{rec}\, A.([a:1] \,;\, B)^{E_B} \\
 & \longrightarrow_{\mathsf{rw}} & \mathrm{rec}\, A.[a:1]^{E_B} \,;\, B^{E_B} \\
 & \longrightarrow_{\mathsf{rw}} & \mathrm{rec}\, A.[a:1] \,;\, B^{E_B} \\
 & \longrightarrow_{\mathsf{rw}} & \mathrm{rec}\, A.[a:1] \,;\, \mathrm{rec}\, B.([b:1] \,;\, A)^{\emptyset} \\
 & \longrightarrow_{\mathsf{rw}} & \mathrm{rec}\, A.[a:1] \,;\, \mathrm{rec}\, B.[b:1]^{\emptyset} \,;\, A^{\emptyset} \\
 & \longrightarrow_{\mathsf{rw}} & \mathrm{rec}\, A.[a:1] \,;\, \mathrm{rec}\, B.[b:1] \,;\, A^{\emptyset} \\
 & \longrightarrow_{\mathsf{rw}} & \mathrm{rec}\, A.[a:1] \,;\, \mathrm{rec}\, B.[b:1] \,;\, A
\end{array}$$

□

3.5.2 Informal Semantics

Each process term represents a potential process which, when given a *context* (i.e., a network and a data environment), is capable of exhibiting some behaviour. We give an informal introduction to the behaviour of process terms in the remainder of this section. The formal semantics is deferred to §3.6.

Send The term, $k!i.x$, denotes a process which causes a message to be queued for transmission on channel k. The message consists of the message identifier, i, and the data value associated with the variable, x. Sending is asynchronous. The process $k!i.x$ cannot be delayed. It causes its message to be queued instantaneously and terminates immediately.

Receive $k?i.x$ is a process which waits to accept a message from channel k. It will only accept a message with the identifier i. It ignores messages with any other identifier, simply allowing time to pass and other network activity to occur. When an i-message reaches its acceptance point during transmission on channel k, then $k?i.x$ must accept the message instantly, causing the data variable x to become associated with the message's data value. $k?i.x$ then terminates immediately.

Compute $[\omega : t_1, t_2]$ is a process which transforms the data state according to the specification of the operation ω. It begins execution immediately and is guaranteed to terminate no later (resp. no sooner) than t_2 (resp. t_1) time units after it has started. The specified change to the data state occurs in a single, instantaneous action at the moment of termination. Two forms of the compute process are deemed important enough to warrant giving them their own names: the `idle` process, defined as $[ID : \infty]$, which never performs any action but simply allows time to pass forever; and the `null` process, defined as $[ID : 0]$, which terminates immediately, leaving the data state unchanged.

Evaluate Guard $\gamma \rightarrow P$ causes the evaluation of the guard γ (which is a predicate on data states) in the current environment. If the guard is satisfied, the process carries on immediately to behave as P; otherwise $\gamma \rightarrow P$ simply idles, allowing time to pass and network activity to occur.

Sequential Composition $P\,;\,Q$ behaves just as P until P terminates. It then carries on immediately to behave as Q, using the state of the network and the data environment at P's termination. If P does not terminate then Q is never started.

Choice $P + Q$ behaves either as P or as Q. The choice is resolved in favour of whichever process can first perform an action. If both P and Q can perform an action simultaneously, the choice is resolved arbitrarily in favour of one of them. Network activity and the passage of time must be allowed by both P and Q in order to occur; neither resolves the choice. The choice operator can be used quite simply to model a timeout, e.g.,

$$k?i.x\;;\;P + [\mathit{Timeout} : 4]\;;\;Q$$

denotes a process which is able to receive a message with identifier i from channel k and then behave like process P, or, alternatively, may execute the *Timeout* operation at time 4 and then behave like process Q. Notice that if an i message becomes available for reception *before* time 4 then it *will be received* and the timeout branch will be discarded. On the other hand, if an i message does not become available at any time up to, and including, time 4 then the timeout branch *will be taken* and the possibility of the message reception will be discarded. If an i message becomes available at *exactly* time 4 then one or the other branch will be taken non-deterministically, i.e., the view is taken that when two or more actions are possible at the same moment in time, we cannot determine the order in which they may occur but must consider all possible interleavings.

Interrupt $P\,[>\,Q$ behaves as P until either Q performs an action or P terminates. In the first case, the system carries on to behave as Q with whatever is the current state of the network and data environment (P is aborted); in the second case, the whole process, $P\,[>\,Q$, terminates. If both P and Q can perform an action simultaneously, the choice is resolved arbitrarily in favour of one of them. Network activity and the passage of time both require the willingness of P and Q to allow them to occur. When time passes, it does so in both P and Q. An interrupt is forced when Q can perform an action but cannot allow time progress, and, at the same time, neither P nor the network can perform any action. In effect, the interrupt operator behaves just like the choice operator, except that the occurrence of an action a in the left operand does not cause the right operand to be discarded unless a is a terminating action.

Parallel Composition The parallel operator, $P\,|\,Q$, gives a simple interleaving of the actions of P and Q. As with the other operators, network activity and the passage of time require the willingness of both P and Q to allow them to occur.

Recursion The process $\mathrm{rec}\, X.P$ denotes a *recursive* process which has the potential for repetitive behaviour.

3.6 Formal system model

The formal model of an embedded control system, of the sort which was introduced informally in §3.2, is given by a tuple (P, N, D), where P is a process term describing the behaviour of the system processes, N is a network consisting of one or more communication channels, and D is a data environment. There are some obvious 'sanity' properties which a model (P, N, D) must satisfy in order to be considered *well-formed*. This section specifies what it means for a model to be well-formed. A well-formed model is called a *bCANDLE system*. A formal semantics is given to a *bCANDLE* system (P, N, D) in a standard way, using structured operational rules in the style of Plotkin [Plo81]. A strong equivalence is defined for *bCANDLE* systems and some simple equational laws are identified.

3.6.1 Well-formed systems

Clearly, there are some models (P, N, D) to which we need not attempt to give a semantics, e.g., if $k!i.x$ is a sub-term of P and either k does not identify a channel in N or $D.x$ is undefined. We rule out such models by defining the set of well-formed models, which we call *bCANDLE systems*. Essentially, a model (P, N, D) is well-formed iff P, N and D agree on their channel identifiers, messages identifiers, data variables, operation names and predicate names, and it is both *transmit-safe* and *receive-safe*. A model (P, N, D) is transmit-safe iff any message which may be sent by P on some channel k is a transmissible message for k in the network N. A model is receive-safe iff, for any message $i.v$ which may be received by P into a data variable x, v is in the type of x.

The send and receive sub-terms of a term P are defined simply.

Definition 3.22 ($\mathsf{snd}(P), \mathsf{rcv}(P)$) Let $P \in \mathit{Proc}^+$ and $\bowtie \in \{;, +, [>, |\}$. The send and receive sub-terms of P are given by $\mathsf{snd}(P)$ and $\mathsf{rcv}(P)$ respectively, which are defined as the least sets satisfying:

$$
\begin{aligned}
\mathsf{snd}(k!i.x) &= \{k!i.x\} & \mathsf{rcv}(k!i.x) &= \emptyset \\
\mathsf{snd}(k?i.x) &= \emptyset & \mathsf{rcv}(k?i.x) &= \{k?i.x\} \\
\mathsf{snd}([\omega : t_1, t_2]) &= \emptyset & \mathsf{rcv}([\omega : t_1, t_2]) &= \emptyset \\
\mathsf{snd}(\gamma \rightarrow P) &= \mathsf{snd}(P) & \mathsf{rcv}(\gamma \rightarrow P) &= \mathsf{rcv}(P) \\
\mathsf{snd}(X) &= \emptyset & \mathsf{rcv}(X) &= \emptyset \\
\mathsf{snd}(\mathrm{rec}\, X.P) &= \mathsf{snd}(P) & \mathsf{rcv}(\mathrm{rec}\, X.P) &= \mathsf{rcv}(P) \\
\mathsf{snd}(P \bowtie Q) &= \mathsf{snd}(P) \cup \mathsf{snd}(Q) & \mathsf{rcv}(P \bowtie Q) &= \mathsf{rcv}(P) \cup \mathsf{rcv}(Q)
\end{aligned}
$$

□

We are now in a position to define the set of *bCANDLE* systems.

Definition 3.23 (*bCANDLE* system) Let K and I be finite sets of channel identifiers and message identifiers. Let Var, Ω and Γ be finite sets of variable names, operation names and predicate names, respectively. Let V be the set of data values. The set of *bCANDLE systems* over $K, I, \mathit{Var}, \Omega$ and Γ is denoted $bCAN_{K,I,\mathit{Var},\Omega,\Gamma}$ and a tuple $(P, N, D) \in bCAN_{K,I,\mathit{Var},\Omega,\Gamma}$ iff $P \in \mathit{Proc}_{K,I,\mathit{Var},\Omega,\Gamma}$, $N \in \mathit{Network}_{K,I}$, $D \in \mathit{DataEnv}_{\mathit{Var},\Omega,\Gamma}$ and the following two properties are satisfied:

$$k!i.x \in \mathsf{snd}(P) \wedge N_k = (M, \_, \_, \_, \_) \Rightarrow \{i.v \mid v \in D.\mathsf{type}(x)\} \subseteq M \tag{3.3}$$
$$k?i.x \in \mathsf{rcv}(P) \wedge N_k = (M, \_, \_, \_, \_) \Rightarrow \{v \mid i.v \in M\} \subseteq D.\mathsf{type}(x) \tag{3.4}$$

□

Conditions 3.3 and 3.4 express the requirements for transmit-safe and receive-safe models, respectively.

Notation. As usual, the subscript is dropped from $bCAN_{K,I,\mathit{Var},\Omega,\Gamma}$ when it can be inferred from the context.

3.6.2 Operational semantics

The semantics of a *bCANDLE* system is given by a labelled timed transition system. The set of labels consists of the network action labels A_{n}, the time passage labels $\mathbb{R}$, and the process action labels A_{p}, defined as follows.

Definition 3.24 (Process Action Labels) Let K and I be sets of channel and message identifiers, respectively. Let Ω be a set of operation names and Γ a set of predicate names. Let V be the set of data values. The set A_{p} of process action labels over K, I, Ω and Γ, is defined

$$\begin{array}{rcl} A_{\mathsf{p}} & \widehat{=} & \Omega \cup \Gamma \cup \{k!i.v \mid k \in K \wedge i \in I \wedge v \in V\} \\ & \cup & \{k?i.v \mid k \in K \wedge i \in I \wedge v \in V\} \end{array}$$

□

Definition 3.25 (*bCANDLE* semantics) The semantics of a system $B \in bCAN$, is given by the timed transition system $\mathcal{T}\,[\![B]\!] = (\Sigma, \sigma^{\mathcal{I}}, L, \longrightarrow)$ where

- $\Sigma = bCAN$, is the set of states of the system.
- The initial state, $\sigma^{\mathcal{I}}$, is B.
- $L = A_{\mathsf{p}} \cup A_{\mathsf{n}} \cup \mathbb{R}$ is the set of transition labels.

$$\textbf{Snd.1}\ \frac{N_k = (s,u) \wedge v = D.x}{(k!i.x, N, D) \xrightarrow{k!i.v} (\checkmark, N[k := (s, u \hookleftarrow i.v)], D)}$$

$$\textbf{Snd.2}\ \frac{N \xrightarrow{\lambda_{\mathsf{n}}}_{\mathsf{n}} N'}{(k!i.x, N, D) \xrightarrow{\lambda_{\mathsf{n}}} (k!i.x, N', D)}$$

$$\textbf{Snd.3}\ \frac{}{(k!i.x, N, D) \xrightarrow{0} (k!i.x, N, D)}$$

$$\textbf{Rcv.1}\ \frac{N_k = (\uparrow i.v, \_)}{(k?i.x, N, D) \xrightarrow{k?i.v} (\checkmark, N, D[x := v])}$$

$$\textbf{Rcv.2}\ \frac{N \xrightarrow{\lambda_{\mathsf{n}}}_{\mathsf{n}} N' \wedge (N_k \neq (\uparrow i.\_, \_) \vee N_k = N'_k)}{(k?i.x, N, D) \xrightarrow{\lambda_{\mathsf{n}}} (k?i.x, N', D)}$$

$$\textbf{Rcv.3}\ \frac{N \xrightarrow{t}_{\mathsf{n}} N'}{(k?i.x, N, D) \xrightarrow{t} (k?i.x, N', D)}$$

$$\textbf{Comp.1}\ \frac{D \xrightarrow{\omega}_{\mathsf{d}} D'}{([\omega : 0, \_], N, D) \xrightarrow{\omega} (\checkmark, N, D')}$$

$$\textbf{Comp.2}\ \frac{N \xrightarrow{\lambda_{\mathsf{n}}}_{\mathsf{n}} N'}{([\omega : t_1, t_2], N, D) \xrightarrow{\lambda_{\mathsf{n}}} ([\omega : t_1, t_2], N', D)}$$

$$\textbf{Comp.3}\ \frac{N \xrightarrow{t}_{\mathsf{n}} N' \wedge t \leq t_2}{([\omega : t_1, t_2], N, D) \xrightarrow{t} ([\omega : t_1 \dot{-} t, t_2 \dot{-} t], N', D)}$$

Fig. 3.5: Rules for Basic Systems

- $\longrightarrow \subseteq (\Sigma \times L \times \Sigma)$ is the least relation which is closed under the structured operational rules of Figures 3.3, 3.5, 3.6 and 3.7. These rules make use of generic labels $\lambda_{\mathsf{p}}, \lambda_{\mathsf{n}}, \lambda_{\mathsf{nt}}$ and λ where λ_{p} ranges over A_{p}, λ_{n} ranges over A_{n}, λ_{nt} ranges over $A_{\mathsf{n}} \cup \mathbb{R}$ and λ ranges over L. We assume that t ranges over $\mathbb{R}$, whereas t_1, t_2 range over $\mathbb{R}_\infty$. $\square$

Termination

The distinguished process name ✓ is used in the semantic rules to indicate successful termination. It is used only in giving the semantics and is not available to a user of the language as a process term. This side-steps many of the tricky issues which arise in attempting to give a proper treatment of termination in a timed setting and permits a standard approach to be taken to the definition of the operational semantics and strong bisimilarity. The reader who is interested

$$\textbf{Gu.1}\ \frac{D \models \gamma}{(\gamma \to P, N, D) \xrightarrow{\gamma} (P, N, D)}$$

$$\textbf{Gu.2}\ \frac{N \xrightarrow{\lambda_{\mathsf{n}}}_{\mathsf{n}} N'}{(\gamma \to P, N, D) \xrightarrow{\lambda_{\mathsf{n}}} (\gamma \to P, N', D)}$$

$$\textbf{Gu.3}\ \frac{N \xrightarrow{t}_{\mathsf{n}} N' \wedge (D \not\models \gamma \vee t = 0)}{(\gamma \to P, N, D) \xrightarrow{t} (\gamma \to P, N', D)}$$

$$\textbf{Seq.1}\ \frac{(P, N, D) \xrightarrow{\lambda} (P', N', D') \wedge P' \not\equiv \checkmark}{(P\ ;\ Q, N, D) \xrightarrow{\lambda} (P'\ ;\ Q, N', D')}$$

$$\textbf{Seq.2}\ \frac{(P, N, D) \xrightarrow{\lambda_{\mathsf{p}}} (\checkmark, N', D')}{(P\ ;\ Q, N, D) \xrightarrow{\lambda_{\mathsf{p}}} (Q, N', D')}$$

$$\textbf{Ch.1}\ \frac{(P, N, D) \xrightarrow{\lambda_{\mathsf{p}}} (P', N', D')}{(P + Q, N, D) \xrightarrow{\lambda_{\mathsf{p}}} (P', N', D')}$$

$$\textbf{Ch.2}\ \frac{(Q, N, D) \xrightarrow{\lambda_{\mathsf{p}}} (Q', N', D')}{(P + Q, N, D) \xrightarrow{\lambda_{\mathsf{p}}} (Q', N', D')}$$

$$\textbf{Ch.3}\ \frac{(P, N, D) \xrightarrow{\lambda_{\mathsf{nt}}} (P', N', D) \wedge (Q, N, D) \xrightarrow{\lambda_{\mathsf{nt}}} (Q', N', D)}{(P + Q, N, D) \xrightarrow{\lambda_{\mathsf{nt}}} (P' + Q', N', D)}$$

$$\textbf{Rec}\ \frac{(P[\mathrm{rec}\, X.P/X], N, D) \xrightarrow{\lambda} (P', N', D')}{(\mathrm{rec}\, X.P, N, D) \xrightarrow{\lambda} (P', N', D')}$$

Fig. 3.6: Rules for Guard, Sequential Composition, Choice and Recursion

in some of the issues which arise when a more satisfying algebraic approach is attempted is referred to [BV97, Ver97a, BR00].

Proposition 3.8 (Time determinism, time additivity) *For any bCANDLE system, B, $\mathcal{T}[\![B]\!]$ is a timed transition system.*

Proof By Definition 2.6 and induction on the depth of the inferences which justify the time transitions. □

3.6.3 Strong equivalence

We define a strong equivalence for *bCANDLE* systems using *bisimulation* [Mil89]. The standard notion of bisimulation is adapted by requiring that bisimilar states have identical contexts, i.e., their networks and data environments must be the

$$\textbf{Int.1}\ \frac{(P,N,D)\xrightarrow{\lambda_{\mathsf{p}}}(P',N',D') \wedge P' \not\equiv \checkmark}{(P\ [>\ Q,N,D)\xrightarrow{\lambda_{\mathsf{p}}}(P'\ [>\ Q,N',D')}$$

$$\textbf{Int.2}\ \frac{(P,N,D)\xrightarrow{\lambda_{\mathsf{p}}}(\checkmark,N',D')}{(P\ [>\ Q,N,D)\xrightarrow{\lambda_{\mathsf{p}}}(\checkmark,N',D')}$$

$$\textbf{Int.3}\ \frac{(Q,N,D)\xrightarrow{\lambda_{\mathsf{p}}}(Q',N',D')}{(P\ [>\ Q,N,D)\xrightarrow{\lambda_{\mathsf{p}}}(Q',N',D')}$$

$$\textbf{Int.4}\ \frac{(P,N,D)\xrightarrow{\lambda_{\mathsf{nt}}}(P',N',D) \wedge (Q,N,D)\xrightarrow{\lambda_{\mathsf{nt}}}(Q',N',D)}{(P\ [>\ Q,N,D)\xrightarrow{\lambda_{\mathsf{nt}}}(P'\ [>\ Q',N',D)}$$

$$\textbf{Par.1}\ \frac{(P,N,D)\xrightarrow{\lambda_{\mathsf{p}}}(P',N',D') \wedge P' \not\equiv \checkmark}{(P \mid Q,N,D)\xrightarrow{\lambda_{\mathsf{p}}}(P' \mid Q,N',D')}$$

$$\textbf{Par.2}\ \frac{(P,N,D)\xrightarrow{\lambda_{\mathsf{p}}}(\checkmark,N',D')}{(P \mid Q,N,D)\xrightarrow{\lambda_{\mathsf{p}}}(Q,N',D')}$$

$$\textbf{Par.3}\ \frac{(Q,N,D)\xrightarrow{\lambda_{\mathsf{p}}}(Q',N',D') \wedge Q' \not\equiv \checkmark}{(P \mid Q,N,D)\xrightarrow{\lambda_{\mathsf{p}}}(P \mid Q',N',D')}$$

$$\textbf{Par.4}\ \frac{(Q,N,D)\xrightarrow{\lambda_{\mathsf{p}}}(\checkmark,N',D')}{(P \mid Q,N,D)\xrightarrow{\lambda_{\mathsf{p}}}(P,N',D')}$$

$$\textbf{Par.5}\ \frac{(P,N,D)\xrightarrow{\lambda_{\mathsf{nt}}}(P',N',D) \wedge (Q,N,D)\xrightarrow{\lambda_{\mathsf{nt}}}(Q',N',D)}{(P \mid Q,N,D)\xrightarrow{\lambda_{\mathsf{nt}}}(P' \mid Q',N',D)}$$

Fig. 3.7: Rules for Interrupt and Parallel Composition

same. This seems essential intuitively and accords with other treatments of bisimulation for systems whose states contain an explicit context [GP94, BV94].

Firstly, the standard definition of bisimulation is given with respect to an equivalence relation between states.

Definition 3.26 (Strong Bisimulation) Let $\mathcal{S} = (\Sigma, \sigma^{\mathcal{I}}, L, \longrightarrow)$ be a LTS. Let $\approx$ be an equivalence relation on Σ. A binary relation $R \subseteq \Sigma \times \Sigma$ is a strong $\approx$-*bisimulation* if $\sigma_1 R \sigma_2$ implies

1. $\sigma_1 \approx \sigma_2$
2. for all $\lambda \in L$, if $\sigma_1 \xrightarrow{\lambda} \sigma_1'$, then $\sigma_2 \xrightarrow{\lambda} \sigma_2'$ for some σ_2' such that $\sigma_1' R \sigma_2'$
3. for all $\lambda \in L$, if $\sigma_2 \xrightarrow{\lambda} \sigma_2'$, then $\sigma_1 \xrightarrow{\lambda} \sigma_1'$ for some σ_1' such that $\sigma_1' R \sigma_2'$ $\square$

Notice that one obtains the standard definition of strong bisimulation by taking $\approx$ to be $\Sigma \times \Sigma$.

Definition 3.27 (Strong equivalence) Let $\mathcal{S} = (\Sigma, \sigma^{\mathcal{I}}, L, \longrightarrow)$ be a LTS. σ_1, σ_2 in Σ are *strongly equivalent* ($\approx$-bisimilar), denoted $\sigma_1 \underline{\leftrightarrow}^{\approx} \sigma_2$, iff there is a strong $\approx$-bisimulation R such that $\sigma_1 R \sigma_2$. $\square$

Strong equivalence is extended to transition systems, as follows.

Definition 3.28 Let $\mathcal{S}_1 = (\Sigma_1, \sigma_1^{\mathcal{I}}, L_1, \longrightarrow_1)$ and $\mathcal{S}_2 = (\Sigma_2, \sigma_2^{\mathcal{I}}, L_2, \longrightarrow_2)$ be LTS's. A $\approx$-*bisimulation* between $\mathcal{S}_1$ and $\mathcal{S}_2$ is a binary relation $R \subseteq \Sigma_1 \times \Sigma_2$, satisfying $\sigma_1^{\mathcal{I}} R \sigma_2^{\mathcal{I}}$ and the three clauses of Definition 3.26. $\mathcal{S}_1$ is *strongly equivalent* ($\approx$-bisimilar) to $\mathcal{S}_2$, denoted $\mathcal{S}_1 \underline{\leftrightarrow}^{\approx} \mathcal{S}_2$, iff there is a $\approx$-bisimulation between them. $\square$

Notation. We write $\sigma_1 \underline{\leftrightarrow} \sigma_2$ for $\sigma_1 \underline{\leftrightarrow}^{\approx} \sigma_2$, and $\mathcal{S}_1 \underline{\leftrightarrow} \mathcal{S}_2$ for $\mathcal{S}_1 \underline{\leftrightarrow}^{\approx} \mathcal{S}_2$, when the relation $\approx$ is clear from the context.

In order to develop a notion of strong equivalence for *bCANDLE* systems, we define *context equivalence* which simply requires that networks and data environments are identical.

Definition 3.29 (Context equivalence) Let $\sigma_1 = (P_1, N_1, D_1)$ and $\sigma_2 = (P_2, N_2, D_2)$ be two *bCANDLE* system states in $bCAN$. σ_1 is *context equivalent* to σ_2, denoted $\sigma_1 \approx_{ND} \sigma_2$, iff $N_1 = N_2$ and $D_1 = D_2$. $\square$

Clearly, $\approx_{ND}$ is an equivalence relation. Now, strong equivalence for *bCANDLE* systems is defined simply as $\approx_{ND}$-bisimilarity.

Definition 3.30 Let $B_1, B_2 \in bCAN$. B_1 is *strongly equivalent* to B_2, denoted $B_1 \underline{\leftrightarrow} B_2$ iff their transition systems are $\approx_{ND}$-bisimilar, i.e. $\mathcal{T}[\![B_1]\!] \underline{\leftrightarrow}^{\approx_{ND}} \mathcal{T}[\![B_2]\!]$ $\square$

We also extend the notion of strong equivalence to the set *Proc* of closed, guarded process terms.

Definition 3.31 Let $P, Q \in Proc$ be closed, guarded process terms. P is *strongly equivalent* to Q, denoted $P \underline{\leftrightarrow} Q$, iff $(P, N, D) \underline{\leftrightarrow} (Q, N, D)$ for all *bCANDLE* systems $(P, N, D), (Q, N, D) \in bCAN$. $\square$

Notation. $bCAN \models P \underline{\leftrightarrow} Q$ denotes the fact that P is strongly equivalent to Q with respect to the semantics of $bCAN$.

Proposition 3.9 (Congruence) *The relation $\underline{\leftrightarrow}$ between terms in Proc, is a congruence with respect to all process operators.*

+.1	$P + Q = Q + P$
+.2	$P + (Q + R) = (P + Q) + R$
+.3	$P + P = P$
+.4	$P + \mathrm{idle} = P$
;.1	$(P \, ; \, Q) \, ; \, R = P \, ; \, (Q \, ; \, R)$
;.2	$(P + Q) \, ; \, R = P \, ; \, R + Q \, ; \, R$
;.3	$\mathrm{idle} \, ; \, P = \mathrm{idle}$
[>.1	$\mathrm{idle} \, [> P = P$
[>.2	$P \, [> \mathrm{idle} = P$
[>.3	$(P \, [> Q) \, [> R = P \, [> (Q \, [> R)$
|.1	$P \mid Q = Q \mid P$
|.2	$P \mid (Q \mid R) = (P \mid Q) \mid R$
|.3	$P \mid \mathrm{idle} = P \qquad \text{if } P \text{ is persistent}$
rec .1	$\mathrm{rec}\, X.P = P[\mathrm{rec}\, X.P/X]$
rec .2	If $P[Q/X] = Q$ and X is guarded in P then $\mathrm{rec}\, X.P = Q$

Tab. 3.2: Equational laws

Proof The structured operational rules which define the semantics fall within the Super-SOS format of [BV94]. The proof of the proposition then follows from the fact that strong equivalence is a congruence with respect to any set of operators defined by such a set of rules. □

3.6.4 Equational laws

Our primary approach to verification is via graph-based exploration techniques such as model checking and reachability analysis, rather than by algebraic reasoning. However, we observe that many of the usual laws are satisfied. Table 3.2 summarises the known laws for *bCANDLE* systems. Consideration of a framework in which it is possible to derive a complete axiomatisation, and useful theorems such as an expansion theorem, is of interest but has been judged, so far, to be of secondary importance to the development of algorithmic analysis techniques.

Notation. We write $bCAN \vdash P = Q$, if P and Q are equivalent modulo the laws of Table 3.2.

Proposition 3.10 (Soundness) *For all process terms $P, Q \in Proc$,*
$$bCAN \vdash P = Q \Rightarrow bCAN \models P \underline{\leftrightarrow} Q$$

Proof The proof is a standard application of bisimulation. □

Notice that the law |.3 holds only for *persistent* systems, where a *bCANDLE* system is regarded as persistent if it cannot reach a state of the form $(\checkmark, N, D)$. For example, $(k!i.x \, ; \, \mathrm{idle}, N, D)$ is persistent, whereas $(k!i.x, N, D)$ is not persistent, for any N and D.

Definition 3.32 (Persistent *bCANDLE* system)
A *bCANDLE* system (P, N, D) is said to be *persistent* if there is no transition sequence $(P, N, D) \longrightarrow^{*} (\checkmark, N'D')$, for any network N' and data environment D'. A process term P is said to be persistent if every *bCANDLE* system (P, N, D) is persistent. $\square$

It is apparent that the law $P \,|\, \text{idle} = P$ does not hold unless P is persistent. Consider, for example, the systems $(\text{null} \,|\, \text{idle}, N, D)$ and (null, N, D), for any N and D. The former has a transition $(\text{null} \,|\, \text{idle}, N, D) \xrightarrow{ID} (\text{idle}, N, D)$ (by **Comp.1** and **Par.2**), while the latter has a transition $(\text{null}, N, D) \xrightarrow{ID} (\checkmark, N, D)$ (by **Comp.1**). It is clear that $(\text{idle}, N, D) \not\leftrightarrow (\checkmark, N, D)$.

It is not difficult to see that for any non-persistent system (P, N, D), there is a persistent system (P', N, D) which has the same behaviour up to the point when P becomes $\checkmark$ and which, thereafter, only allows the completion of network activity and the progress of time. For example, we can take P' to be either P ; idle or $P \,|\, \text{idle}$.

From now on we assume that we are only dealing with persistent systems, unless stated otherwise.

3.7 A simple example

In this section the use of *bCANDLE* is illustrated by modelling the simple flow control system which was introduced in §1.1. The purpose of the system is to maintain the flow of liquid through a pipe at a preset constant rate. Assume that the system is implemented using two distributed processors: one for reading the flow sensor, the other for adjusting the control valve. The processors are connected by a CAN bus operating at 1Mbit/sec. Figure 3.8 shows a *bCANDLE* model of the system. It consists of two processes: *Flow* and *Valve*.

Flow models a process which periodically reads a flow sensor and broadcasts its value in a *flow* message. It is assumed that the implementation requires between 85 and 90 μsecs to sample the flow sensor, condition the signal and configure a CAN controller to transmit the *flow* message. A hardware timer, which implements the periodic behaviour of the process, interrupts at intervals of approximately 10 msecs.

Valve models a process which repeatedly waits to receive a *flow* message, executes a control algorithm to calculate a new value for the valve position, and instructs an actuator to move the valve to its new position. It is assumed that it takes between 200 to 300 μsecs from receipt of a *flow* message to the configuration of the valve actuator.

The `network` section of the *bCANDLE* model gives the static attributes of the communication channel implemented by the CAN bus. Channel k models a CAN bus which transmits *flow* messages with a transmission latency of between 43 and 53 μsecs from start of transmission to acceptance test, and a latency of between 10 and 12 μsecs from acceptance test to bus idle.

```
Flow | Valve

where

Flow = [ReadSensor:85,90] ; k!flow.x ; idle
           [> [PERIOD:10000,10250] ; Flow

Valve = k?flow.y ; [AdjustValve:200,300]; Valve

network
  /*        pri dlb dub dlB duB   */
  k = (flow : 1, 43, 53, 10, 12)

data x, y
```

Fig. 3.8: Flow regulator in *bCANDLE*

The `data` section introduces the names of data variables used in the model. Initially, the values of all variables are undefined. Variable types, operation specifications and predicate definitions are assumed to be defined externally. Currently, a *bCANDLE* model can be explored using simulators implemented in either Prolog or C. Both simulators require the necessary data definitions to be provided in the host language. In this example, we abstract entirely from the effects of data by assuming that all variables are of type `unit` and all data operations leave the data state unchanged. The modelling of data is discussed in more detail in §6.3.

Our Prolog simulator for *bCANDLE* is a direct implementation of the transition semantics. This approach allows the exploration of *bCANDLE* system models and also helps in gaining confidence in the operational semantics. A similar approach to the animation of the hardware description language VERILOG [Gol96] has been proposed recently by Bowen [Bow99].

A (slightly modified) simulator trace of the flow regulator example is shown in Figure 3.9. The following conventions are adopted:

- A system state is shown as a tuple (P, N) – the data component D is omitted since it is not of interest here.
- The network component N shows only the dynamic attributes of the single channel k.
- The unit value is written as **1**.
- Time delays are chosen arbitrarily from the allowable range of values.

Exploration of a system model in this way can lead quickly to a good understanding of system behaviour. A more thorough exploration can be achieved by applying the model-checking techniques introduced in the next chapter.

$$
\begin{array}{lll}
(\mathit{Flow} \mid \mathit{Valve}, & (\Downarrow, \langle\rangle)) & \xrightarrow{90} \\
(\mathit{Flow}1 \mid \mathit{Valve}, & (\Downarrow, \langle\rangle)) & \xrightarrow{\mathit{ReadSensor}} \\
(\mathit{Flow}2 \mid \mathit{Valve}, & (\Downarrow, \langle\rangle)) & \xrightarrow{k!\mathit{flow}.\mathbf{1}} \\
(\mathit{Flow}3 \mid \mathit{Valve}, & (\Downarrow, \langle \mathit{flow}.\mathbf{1}\rangle)) & \xrightarrow{k \rightsquigarrow \mathit{flow}.\mathbf{1}} \\
(\mathit{Flow}3 \mid \mathit{Valve}, & (\overset{43,53}{\rightsquigarrow} \mathit{flow}.\mathbf{1}, \langle\rangle)) & \xrightarrow{50} \\
(\mathit{Flow}4 \mid \mathit{Valve}, & (\overset{0,3}{\rightsquigarrow} \mathit{flow}.\mathbf{1}, \langle\rangle)) & \xrightarrow{k \Uparrow \mathit{flow}.\mathbf{1}} \\
(\mathit{Flow}4 \mid \mathit{Valve}, & (\uparrow \mathit{flow}.\mathbf{1}, \langle\rangle)) & \xrightarrow{k?\mathit{flow}.\mathbf{1}} \\
(\mathit{Flow}4 \mid \mathit{Valve}1, & (\uparrow \mathit{flow}.\mathbf{1}, \langle\rangle)) & \xrightarrow{\mathit{flow}.\mathbf{1} \rightsquigarrow k} \\
(\mathit{Flow}4 \mid \mathit{Valve}1, & (\mathit{flow}.\mathbf{1} \overset{10,12}{\rightsquigarrow}, \langle\rangle)) & \xrightarrow{10} \\
(\mathit{Flow}5 \mid \mathit{Valve}2, & (\mathit{flow}.\mathbf{1} \overset{0,2}{\rightsquigarrow}, \langle\rangle)) & \xrightarrow{k\Downarrow} \\
(\mathit{Flow}5 \mid \mathit{Valve}2, & (\Downarrow, \langle\rangle)) & \xrightarrow{200} \\
(\mathit{Flow}6 \mid \mathit{Valve}3, & (\Downarrow, \langle\rangle)) & \xrightarrow{\mathit{AdjustValve}} \\
(\mathit{Flow}6 \mid \mathit{Valve}, & (\Downarrow, \langle\rangle)) & \xrightarrow{9700} \\
(\mathit{Flow}7 \mid \mathit{Valve}, & (\Downarrow, \langle\rangle)) & \xrightarrow{\mathit{PERIOD}} \\
(\mathit{Flow} \mid \mathit{Valve}, & (\Downarrow, \langle\rangle)) & \longrightarrow \\
\vdots & &
\end{array}
$$

where the process identifiers are defined as follows:

$$
\begin{array}{rcl}
\mathit{Flow}1 & = & [\mathit{ReadSensor} : 0, 0] \ ; \ k!\mathit{flow}.x \ ; \ \text{idle} \ [> [\mathit{PERIOD} : 9910, 10160] \ ; \ \mathit{Flow} \\
\mathit{Flow}2 & = & k!\mathit{flow}.x \ ; \ \text{idle} \ [> [\mathit{PERIOD} : 9910, 10160] \ ; \ \mathit{Flow} \\
\mathit{Flow}3 & = & \text{idle} \ [> [\mathit{PERIOD} : 9910, 10160] \ ; \ \mathit{Flow} \\
\mathit{Flow}4 & = & \text{idle} \ [> [\mathit{PERIOD} : 9860, 10110] \ ; \ \mathit{Flow} \\
\mathit{Flow}5 & = & \text{idle} \ [> [\mathit{PERIOD} : 9850, 10100] \ ; \ \mathit{Flow} \\
\mathit{Flow}6 & = & \text{idle} \ [> [\mathit{PERIOD} : 9650, 9900] \ ; \ \mathit{Flow} \\
\mathit{Flow}7 & = & \text{idle} \ [> [\mathit{PERIOD} : 0, 200] \ ; \ \mathit{Flow} \\
\mathit{Valve}1 & = & [\mathit{AdjustValve} : 200, 300] \ ; \ \mathit{Valve} \\
\mathit{Valve}2 & = & [\mathit{AdjustValve} : 190, 290] \ ; \ \mathit{Valve} \\
\mathit{Valve}3 & = & [\mathit{AdjustValve} : 0, 90] \ ; \ \mathit{Valve}
\end{array}
$$

Fig. 3.9: Simulator trace of the flow regulator example

3.8 Conclusions and Related Work

The language introduced in this chapter draws on ideas from a variety of sources, mainly in the field of process algebra. Our concern, however, has not been to develop a new process algebra but to design a language with a formal semantics, which is suitable for the pragmatic purpose of modelling a particular class of broadcasting embedded control systems, and which is amenable to analysis by model checking as discussed later in Chapters 4 and 5. Therefore, whenever we have had to choose between an intuitively 'natural' syntax or semantics, on the one hand, and truly satisfying algebraic properties, on the other hand, we have erred in favour of the former. The result is a practical modelling language which accommodates prioritised, CAN-style communication over latent channels, has a dense time semantics, and is amenable to a variety of efficient analysis techniques, known from the study of timed automata.

3.8.1 Broadcast communication and Real-Time

The *bCANDLE* communication mechanism is an *asynchronous*, *broadcast* of messages with *explicit transmission latency*. These characteristics seem to be natural for the performance modelling of CAN-like systems. However, there appears to be no other formal language which combines all three properties in a single communication primitive. We provide a short review of those approaches which seem to come closest to providing what is required.

An early recognition of the importance of broadcast communication is seen in [Geh84], which describes a number of programming examples in a CSP-like language, extended with both unbuffered and buffered broadcast primitives. Some notion of the passing of time is offered by a *delay* construct, but communication is instantaneous and the formal semantics of the language is not considered. The inadequacy of a point-to-point communication primitive for modelling broadcast networks is addressed in [CA91], where a proposal is made for the extension of the formal description technique Estelle [ISO88a] with primitive broadcast channels. The synchronous programming languages, such as ESTEREL [BG92], Lustre [HLR92], Argos [Mar92] and Statecharts [Har87], offer a broadcast primitive, but their reliance on the synchrony hypothesis makes them unsuitable for use in distributed systems. This problem is addressed in [BRS93], which envisages a distributed system as a collection of locally reactive ESTEREL nodes communicating asynchronously with each other. However, the proposed asynchronous communication is the CSP rendezvous, not an asynchronous broadcast. Prasad has developed the Calculus of Broadcasting Systems (CBS) [Pra95], which offers an asynchronous, unbuffered broadcast primitive. A timed version of the calculus is introduced in [Pra96]. His concerns are primarily with algebraic properties of the language, such as an expansion theorem and a complete axiomatisation, rather than with the development of an expressive language for modelling embedded systems. For this reason, there are some features of *bCANDLE* which would be difficult to capture in CBS, e.g., the interrupt operator and transmission latency. In his thesis [Hol94], Holmer discusses the relationship of CBS and SCCS [Mil89] and gives a translation from

CBS to SCCS. This opens the possibility of automated analysis of CBS models using the Concurrency Workbench [CPS93]; however, dense real-time is not addressed in this framework. A language which is closer to *bCANDLE* is the Timed Statechart language of [KP92], which offers an asynchronous broadcast primitive and an expressive, timed process language; however, broadcasts are instantaneous, so modelling of transmission latency requires the introduction of a process to capture the delay in each broadcast channel. A similar approach needs to be adopted to make use of the broadcast primitive introduced into Real-Time CSP in [DJS92]. The importance of broadcast communication as a primitive concept has been recognised again more recently in [EFM99], which addresses the model-checking problems for safety and liveness properties in broadcast protocols; once again, real-time issues are not considered.

3.8.2 Process Operators

Clearly, the process language of *bCANDLE* has been influenced by a number of other languages. Here, we briefly acknowledge our debts.

The relative time-stamped actions, sequential composition and choice operators are as seen in ACP_ρ [BB91], while our parallel composition is the standard, asynchronous, interleaving operator. The interrupt operator has the same semantics as the operator of ET-LOTOS, which allows time to pass in both arguments, rather than the corresponding operator of Real-Time CSP, which allows time to pass only in its left argument [BDS94]. This allows us to omit RT-CSP's watchdog operator, at no cost to expressiveness. Our treatment of state, particularly with respect to the modelling of asynchronous broadcast channels and the associated send and receive operations has been influenced by the work of Kesten and Pnueli [KP92].

4. ANALYSIS VIA TIMED AUTOMATA

In this chapter we define a method for generating timed automata (TA) from *bCANDLE* system descriptions. The method described supports the automatic construction of a TA which is equivalent, in a well-defined sense, to a given *bCANDLE* description. This introduces the possibility of using the powerful verification techniques and tools, described in §2.7, for the analysis of *bCANDLE* systems.

Translation from modelling languages to automata has been studied in a variety of settings. Early approaches were concerned with the family of *synchronous* programming languages which includes ESTEREL [BG92], Lustre [HLR92] and Argos [JM95]. The problem has also been studied for the untimed process algebra LOTOS [Gar92] and for the timed languages ATP [Nic92, NSY92, Yov93], AORTA [BHKR95] and ET-LOTOS [Her98]. This is the first treatment which considers a language which combines latent broadcast communication with data and dense time.

The organisation of the rest of the chapter is as follows. Section 4.1 gives an informal introduction to the objectives of the chapter using a simple example. In §4.2 we revise our system models to include explicit clock variables. This modification facilitates the construction of a TA for a *bCANDLE* description. The construction and its correctness are considered in §4.3. Section 4.4 provides the foundations for the practical implementation of the construction. The application of the method is demonstrated with an example in §4.5 and finally we present our conclusions in §4.6.

4.1 A *bCANDLE* System and its Timed Automaton

We can illustrate our objectives in this chapter with a modified version of the flow regulator example (§3.7). The example is curtailed in order to simplify the presentation.

Consider the *bCANDLE* description shown in Figure 4.1. The example models a *one-shot* flow regulator, in which a single interaction occurs between a flow sensor and a valve controller. The system consists of two processes, *Flow* and *Valve*, and a broadcast channel k. The *Flow* process takes a single reading from a flow sensor. We abstract from the actual value read by the *ReadSensor* operation. *Flow* broadcasts the *flow* message on channel k and then idles forever. We assume that k can transmit only one type of message, namely *flow* messages, and that it does so within the bounds shown in its declaration in the network section. The *Valve* process waits to receive the flow reading from channel k. When the message is received, *Valve* executes its *AdjustValve*

```
Flow | Valve

where

Flow = [ReadSensor:85,90] ; k!flow.x ; idle

Valve = k?flow.y ; [AdjustValve:200,300]; idle

network
  /*        pri dlb dub dlB duB   */
  k = (flow : 1, 43, 53, 10, 12)

data x, y
```

Fig. 4.1: One-shot flow regulator in *bCANDLE*

operation and then also idles forever.

An equivalent behaviour can be expressed using the TA of Figure 4.2. We recall from §2.5.4 that a TA is a tuple $\mathcal{A} = (Q, q^{\mathcal{I}}, A, \mathcal{H}, E, I)$ of locations, initial location, action labels, clocks, edges and invariant function. The example automaton has eleven locations, twelve edges and four clocks. The initial location is (0). The set of TA labels is the set comprising the network action labels and process action labels of the *bCANDLE* system. Now consider the behaviour of the TA. Clock $H4$ is active in location (0). It constrains the control in the automaton to reside in this location for not more than 90 time units. After 85 time units the *ReadSensor* transition can be taken. This captures the behaviour of $[\mathit{ReadSensor} : 85, 90]$ and is typical of the translation of a *bCANDLE* computation. The edge from location (1) to location (2) models the instantaneous queueing of the message *flow* on channel k. The urgency of the action is captured by the invariant condition $H1 \leq 0$ attached to location (1). Edges from (2) to (3) and from (3) to (4) represent the transmission of the message up to the point at which it is available for reception by waiting processes. Notice that clock $H2$ has been allocated to channel k and is used to capture all non-urgent timing constraints on the behaviour of this channel. The edge from (4) to (5) represents the reception of the message by the *Valve* process. Subsequent edges capture the possible interleavings of actions as the channel enters its post-acceptance phase and becomes free, while the *Valve* process completes its *AdjustValve* operation. When control eventually reaches location (10), the system idles forever.

It is not difficult to convince oneself that the TA describes the same system as the *bCANDLE* model. Notice, however, that some of the edges in the TA are redundant. For example, the edge between locations (5) and (6) has a guard, $H3 \geq 200$, which can never be satisfied, since both $H1$ and $H3$ are reset on entry to location (5), and the invariant at (5) requires $H1 \leq 0$. Similarly, the guard on the edge between locations (7) and (8) is unsatisfiable, since $H2$ and $H3$ must be 0 on entry to location (7). The inclusion of such redundant edges does not compromise the equivalence between the TA and the *bCANDLE*

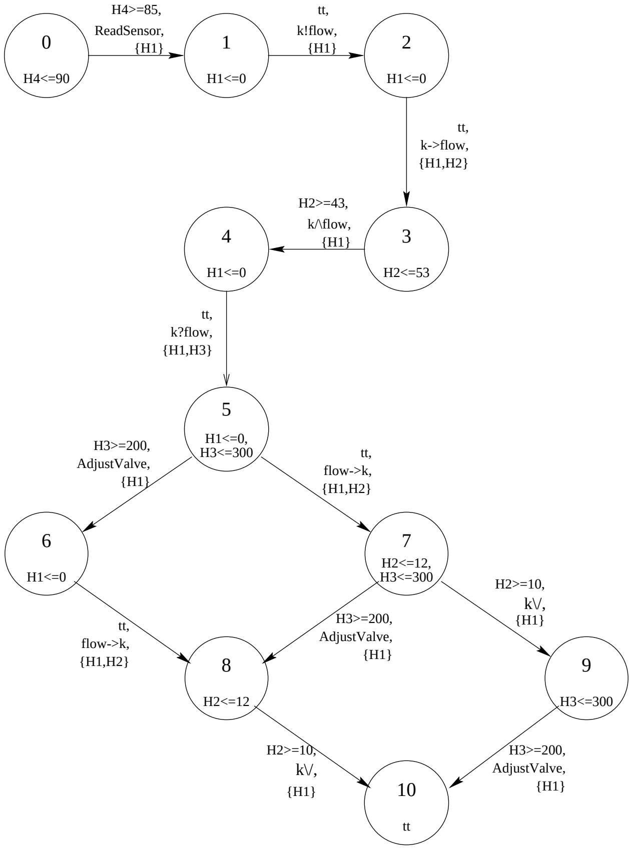


Fig. 4.2: A timed automaton for the one-shot flow regulator

model, but the efficiency of automatic analysis procedures based on the TA may be degraded. This problem is addressed in Chapter 5.

The purpose of the remainder of this chapter is to define a translation from *bCANDLE* models to their equivalent TA, and to show how the translation can be implemented efficiently.

4.2 Models with explicit clocks

Timed automata model the passing of time by using explicit clock variables. On the other hand, timed process algebra represent either absolute or relative time in the syntax of process terms, without the use of explicit clock variables. This is the case for *bCANDLE*. As a first step in the translation from *bCANDLE* to timed automata, explicit clock variables are introduced into *bCANDLE* models. The approach is similar to that adopted in the translation of ATP [Nic92].

As an example, look again at the one-shot flow regulator of Figure 4.1. Its TA is constructed on the assumption that the process term has been decorated with clock variables as follows:

$$\begin{array}{l}[\mathit{ReadSensor} : 85, 90]^{H4};\ k!\mathit{flow}.x;\ \text{idle} \\ | \\ k?\mathit{flow}.y;\ [\mathit{AdjustValve} : 200, 300]^{H3};\ \text{idle}\end{array}$$

Similarly, it is assumed that the network channel k has been decorated with the clock variable $H2$. The clock $H1$ is reserved to enforce *urgent* actions, such as $k!i.x$, which must be either executed or disabled without delay. Now, imagine that time advances by 10 time units from the initial state, and consider the effect of this time passage on the term $[\mathit{ReadSensor} : 85, 90]$. In an unclocked scenario, we expect to see this term evolve to $[\mathit{ReadSensor} : 75, 80]$. However, when using explicit clock variables, we find that $[\mathit{ReadSensor} : 85, 90]$ remains unchanged but the value of clock $H4$ advances from 0 to 10. In this case, a *ReadSensor* transition becomes enabled when the value of $H4$ reaches 85. Network transitions are controlled similarly by clock $H2$.

The remainder of this section formalises this idea by introducing the basic definitions of explicitly clocked *bCANDLE* models. Throughout, it is assumed that $\mathcal{H}$ is the set of clock variables and h ranges over $\mathcal{H}$.

4.2.1 Clocked Networks

In the case of the network model, each network channel is simply associated with a clock variable which is used to measure the passage of time during message transmission.

Let K be a finite set of channel identifiers and I a finite set of message identifiers.

Definition 4.1 (Clocked Network) A *clocked network* over K and I is a mapping $\widehat{N} : K \to \mathit{Channel}_I \times \mathcal{H}$. The set of clocked networks over K and I is denoted $\widehat{\mathit{Network}}_{K,I}$, where $\widehat{\mathit{Network}}_{K,I} \mathrel{\widehat{=}} K \to \mathit{Channel}_I \times \mathcal{H}$. □

Remark 4.1 Recall that the constants occurring in the clock constraints of the invariant function and edges of a TA are required to be natural numbers (§2.5.3). Therefore, it is necessary to restrict attention to clocked networks in which the transmission latency function of every channel is defined by a function $\delta : M \to \mathbb{N}_\infty \times \mathbb{N}_\infty \times \mathbb{N}_\infty \times \mathbb{N}_\infty$, where $\mathbb{N}_\infty \mathrel{\widehat{=}} \mathbb{N} \cup \{\infty\}$ (cf. Definition 3.4). We require that all clocked networks $\widehat{N} \in \widehat{\mathit{Network}}$ satisfy this constraint. □

Notation. Let $\widehat{N}$ be a clocked network and $\widehat{N}_k = (\eta, h)$. The notation η^h is used as an abbreviation for (η, h). In fact, we sometimes omit the clock variable entirely when we do not intend to refer to it in some context, and simply write $\widehat{N}_k = \eta$.

Definition 4.2 (Clock Variables: Network) Let K be a set of channel identifiers. The clock variables used in a clocked network are given by the function

$\mathsf{clk} : \widehat{Network}_K \to 2^{\mathcal{H}}$, where $\mathsf{clk}(\widehat{N}) \mathrel{\widehat{=}} \{h \mid k \in K \wedge \widehat{N}_k = (\_, h)\}$ □

Definition 4.3 (Unclocked Network) The *unclocked network* corresponding to a clocked network is given by the function $\mathsf{unclk} : \widehat{Network}_K \to \mathit{Network}$, where $\mathsf{unclk}(\widehat{N}) \mathrel{\widehat{=}} \{k \mapsto \eta \mid k \in K \wedge \widehat{N}_k = (\eta, \_)\}$. □

4.2.2 Clocked Process Terms

In defining the set of clocked process terms, we also introduce a number of syntactic restrictions which ensure that a TA can be constructed in a straightforward manner:

1. Constants t_1, t_2 in time-bounded computations $[\omega : t_1, t_2]$ are required to be natural numbers. This is for similar reasons to those discussed in Remark 4.1.
2. The use of the parallel operator is restricted to the top-level. This restriction simplifies the implementation of the TA translation.
3. All terms are required to have *static control*. This is discussed in more detail below.

In practice, these restrictions do not severely curtail the models which can be expressed. In fact, we will see that the high-level language *CANDLE* allows the expression of a wide variety of systems, and yet all *CANDLE* programs can be translated into *bCANDLE* models which satisfy these syntactic constraints.

The first two restrictions are captured in the following definition of clocked process terms.

Definition 4.4 (Clocked Process Terms) The set of clocked process terms, $\widehat{Proc}^+$, over $K, I, \mathit{Var}, \Omega$ and Γ is defined inductively by:

$$\begin{aligned}
\widehat{P} &::= \widehat{Q} \\
&\mid\ \ \widehat{P} \mid \widehat{P} \\
\widehat{Q} &::= k!i.x \mid k?i.x \mid [\omega : t_1, t_2]^h \mid \gamma \to \widehat{Q} \\
&\mid\ \ \widehat{Q}\ ;\ \widehat{Q} \mid \widehat{Q} + \widehat{Q} \mid \widehat{Q}\ [>\ \widehat{Q} \\
&\mid\ \ \mathrm{rec}\, X.\widehat{Q} \mid X
\end{aligned}$$

where $\mathcal{H}$ is a set of clocks, $h \in \mathcal{H}$, $t_1, t_2 \in \mathbb{N}_\infty$, $\widehat{P}$ ranges over $\widehat{Proc}^+$, $\widehat{Q}$ ranges over the terms in $\widehat{Proc}^+$ except those containing the parallel operator, and the other variables are defined as usual (Definition 3.16). □

In keeping with our previous convention, we use the variables $\widehat{\beta}, \widehat{\beta}', \widehat{\beta}_1$ etc. to range over the *clocked basic terms*, which are of the form $k!i.x$, $k?i.x$ and $[\omega : t_1, t_2]^h$.

The definitions over the structure of terms given in §3.5 are easily extended to clocked terms, and we shall refer to *closed* and *guarded* clocked process terms without further explanation. The set of closed, guarded, clocked process terms is denoted $\widehat{Proc}$.

Static control

There are some *bCANDLE* systems which cannot be represented by any finite TA. For example, consider a system $(P, N, D) \in bCAN$ where the process P is defined as

$$P \mathrel{\widehat{=}} \mathrm{rec}\, X.((([a:0]\ ;\ X)\ [>\ ([b:0]\ ;\ \mathrm{idle})).$$

This can give rise to an unbounded expansion

$$((\mathrm{rec}\, X.[a:0]\ ;\ X\ [>\ [b:0]\ ;\ \mathrm{idle})\ [>\ [b:0]\ ;\ \mathrm{idle})\ [>\ \cdots\ [>\ [b:0]\ ;\ \mathrm{idle}$$

by repeatedly unwinding the recursion. In the general case, an infinite number of locations are required in a TA generated by the translation of systems containing an unbounded expansion of this sort. Similar difficulties can be seen with recursion involving parallel and sequential composition. Clearly, such terms should be excluded from consideration when proposing a translation scheme to finite TA.

Although, it is difficult to provide a precise characterisation of the offending terms, it is possible to identify a larger set which clearly contains all non-finite cases; we call this the set of terms which *compromise static control.* Roughly speaking, a term compromises static control if it contains a recursion through the parallel operator `|` or to the left of the sequential composition or interrupt operators, `;` and `[>`. This idea is stated formally in the following definition:

Definition 4.5 (Static control) A term $P \in Proc$ *compromises static control* if P is of the form $\mathrm{rec}\, X.P_1$ and any of the following conditions hold:

1. P_1 contains a sub-term of the form $Q \mid R$ and $X \in \mathsf{fv}(Q) \cup \mathsf{fv}(R)$;
2. P_1 contains a sub-term of the form $Q\ ;\ R$ and $X \in \mathsf{fv}(Q)$;
3. P_1 contains a sub-term of the form $Q\ [>\ R$ and $X \in \mathsf{fv}(Q)$.

A term $P \in Proc$ *has static control* iff P does not contain any term which compromises static control. A *bCANDLE* system $(P, N, D) \in bCAN$ *has static control* iff P has static control. □

This definition 4.5 is extended naturally to clocked process terms. However, notice that by restricting the use of the parallel operator to the top-level in clocked terms, there is no possibility of static control being compromised by a clocked term satisfying condition (1). The benefits of restricting attention to systems having static control can be summarised as follows [Gar92]:

- A finite TA can be constructed for systems with static control.

- The property of static control is decidable using a simple and efficient algorithm.
- It is easy for the system developer to understand the constraint and to develop models which satisfy it.
- TA construction for systems with static control can be implemented efficiently.
- Most systems of practical interest can be modelled within the required constraint.

Unless stated otherwise, we assume from now on that clocked process terms have static control.

Operations on clocked process terms

There are a number of operations on the syntax of clocked terms which are useful. The functions clk, iclk and unclk are defined below.

Definition 4.6 (Clock Variables) The clock variables of a clocked process term are identified by the function $\mathsf{clk} : \widehat{Proc}^{+} \rightarrow 2^{\mathcal{H}}$, defined as the least set satisfying:

$$\begin{array}{rcl}
\mathsf{clk}([\omega : t_1, t_2]^h) & = & \{h\} \\
\mathsf{clk}(k!i.x) & = & \mathsf{clk}(k?i.x) = \mathsf{clk}(X) = \emptyset \\
\mathsf{clk}(\gamma \rightarrow \widehat{P}_1) & = & \mathsf{clk}(\widehat{P}_1) \\
\mathsf{clk}(\widehat{P}_1 \bowtie \widehat{P}_2) & = & \mathsf{clk}(\widehat{P}_1) \cup \mathsf{clk}(\widehat{P}_2), \quad \bowtie \in \{;, +, [>, |\} \\
\mathsf{clk}(\mathrm{rec}\, X.\widehat{P}_1) & = & \mathsf{clk}(\widehat{P}_1)
\end{array}$$

□

Definition 4.7 (Initial Clock Variables) The *initial* clock variables of a clocked process term, $\widehat{P}$, are identified by the function $\mathsf{iclk} : \widehat{Proc}^{+} \rightarrow 2^{\mathcal{H}}$, defined as the least set satisfying:

$$\begin{array}{rcl}
\mathsf{iclk}([\omega : t_1, t_2]^h) & = & \{h\} \\
\mathsf{iclk}(k!i.x) & = & \mathsf{iclk}(k?i.x) = \emptyset \\
\mathsf{iclk}(\gamma \rightarrow \widehat{P}_1) & = & \emptyset \\
\mathsf{iclk}(\widehat{P}_1 \,;\, \widehat{P}_2) & = & \mathsf{iclk}(\widehat{P}_1) \\
\mathsf{iclk}(\widehat{P}_1 \bowtie \widehat{P}_2) & = & \mathsf{iclk}(\widehat{P}_1) \cup \mathsf{iclk}(\widehat{P}_2), \quad \bowtie \in \{+, [>, |\} \\
\mathsf{iclk}(\mathrm{rec}\, X.\widehat{P}_1) & = & \mathsf{iclk}(\widehat{P}_1[\mathrm{rec}\, X.\widehat{P}_1/X])
\end{array}$$

where $\mathsf{iclk}(\widehat{P})$ is well defined iff $\widehat{P}$ is guarded. □

Definition 4.8 (Unclocked process term) The unclocked process term corresponding to a clocked process term is given by the function $\mathsf{unclk} : \widehat{Proc}^{+} \to Proc^{+}$, where $\mathsf{unclk}(\widehat{P})$ is defined by:

$$\begin{array}{rcl}
\mathsf{unclk}(k!i.x) & \widehat{=} & k!i.x \\
\mathsf{unclk}(k?i.x) & \widehat{=} & k?i.x \\
\mathsf{unclk}([\omega : t_1, t_2]^h) & \widehat{=} & [\omega : t_1, t_2] \\
\mathsf{unclk}(\gamma \to \widehat{P}_1) & \widehat{=} & \gamma \to \mathsf{unclk}(\widehat{P}_1) \\
\mathsf{unclk}(\widehat{P}_1 \bowtie \widehat{P}_2) & \widehat{=} & \mathsf{unclk}(\widehat{P}_1) \bowtie \mathsf{unclk}(\widehat{P}_2), \quad \bowtie \in \{;, +, [>, |\} \\
\mathsf{unclk}(\mathrm{rec}\, X.\widehat{P}_1) & \widehat{=} & \mathrm{rec}\, X.\mathsf{unclk}(\widehat{P}_1) \\
\mathsf{unclk}(X) & \widehat{=} & X
\end{array}$$

□

These operations are illustrated in the following small example.

Example 4.1 Let $\widehat{P}$ be the clocked process term defined by

$$\begin{array}{l}
\widehat{P} \mathrel{\widehat{=}} \mathrm{rec}\, X.([\mathit{Init} : t_1]^{H1} \,;\, k?\mathit{flow}.x \,;\, [\mathit{TestFlow} : t_2]^{H2}; \\
\qquad\qquad (\mathit{FlowOk} \to [\mathit{Delay} : t_3]^{H3} + \mathit{FlowHigh} \to k!\mathit{alarm}.x \,;\, \mathrm{idle}))\,;\, X
\end{array}$$

Then, $\mathsf{clk}(\widehat{P})$ gives the set of all clock variables used in $\widehat{P}$, i.e. $\mathsf{clk}(\widehat{P}) = \{H1, H2, H3\}$. The set $\mathsf{iclk}(\widehat{P}) = \{H1\}$ gives the set of clocks which can influence the initial behaviour of $\widehat{P}$. Finally, the unclocked term $\mathsf{unclk}(\widehat{P})$ is just $\widehat{P}$ with all clock variables removed:

$$\begin{array}{l}
\mathsf{unclk}(\widehat{P}) = \mathrm{rec}\, X.([\mathit{Init} : t_1] \,;\, k?\mathit{flow}.x \,;\, [\mathit{TestFlow} : t_2]; \\
\qquad\qquad (\mathit{FlowOk} \to [\mathit{Delay} : t_3] + \mathit{FlowHigh} \to k!\mathit{alarm}.x \,;\, \mathrm{idle}))\,;\, X
\end{array}$$

□

4.2.3 Safe Clock Allocations

So far, we have imposed no constraints on how clocks can be allocated to process terms and networks. Efficiency suggests that we should use as few clock variables as possible. However, it is clear that some clock allocations will cause problems. For example, consider the clocked term

$$[\mathit{ReadSensor} : 10]^{H1} \,;\, [\mathit{LogData} : 20]^{H1} \,;\, \mathrm{idle} \mid [\mathit{ComputeSetPoint} : 15]^{H1} \,;\, \mathrm{idle}\,.$$

The *ReadSensor* transition should reset $H1$ so that it can be used to measure the progress of *LogData*. On the other hand, if *ReadSensor* resets $H1$, then the passage of time for the *ComputeSetPoint* computation will not be measured properly: the reset of $H1$ at time 10 will delay the execution of *ComputeSetPoint* until time 25, which is clearly not the intended behaviour. Similar difficulties can be observed with clock allocations to network channels.

The problem exists when two or more system components share the use of a clock variable but do not agree on the instants when it should be reset. In this case, we say that the system exhibits *clock (variable) contention*, otherwise it is said to be *contention free*.

In the absence of recursion, we can be sure that a system can never evolve to one which exhibits clock contention if the sets of clock variables allocated to process terms involved in an interrupt or parallel composition are disjoint, and each network channel also has its own distinct clock variable. However, with recursion, even this restriction is not enough to remove the possibility of clock contention.

Example 4.2 Consider the term

$$\widehat{P} \mathrel{\widehat{=}} \mathrm{rec}\, X.[a:2]^{H1}\,;\,([b:1]^{H1}\,;\,[c:2]^{H1}\,[>X).$$

Ignoring network and data environment, we see that $\widehat{P}$ can evolve by the passage of two units of time, and the execution of an a-action, to the term

$$[b:1]^{H1}\,;\,[c:2]^{H1}\,[>(\mathrm{rec}\, X.[a:2]^{H1}\,;\,([b:1]^{H1}\,;\,[c:2]^{H1}\,[>X)).$$

Now, when the b-action is executed after the passage of one further time unit, we see the problem of clock variable contention. On the one hand, clock $H1$ should be reset in order to begin timing the computation $[c:2]^{H1}$, but, on the other hand, $H1$ must not be reset since it is currently required in timing the computation $[a:2]^{H1}$. $\square$

This example prompts us to introduce one final restriction on the syntax of process terms, namely, that in any term of the form $\widehat{P}_1\,[>\widehat{P}_2$, the term $\widehat{P}_2$ must be guarded.
These ideas are summarised by the notion of a *safely clocked process term.*

Definition 4.9 (Safely clocked process term) $\widehat{P} \in \widehat{Proc}$ is said to be *safely clocked* iff all sub-terms $\widehat{P}'$ of $\widehat{P}$ satisfy

1. if $\widehat{P}'$ is of the form $\widehat{P}_1\,[>\widehat{P}_2$, then $\widehat{P}_2$ is guarded, and the initial clock variables of $\widehat{P}_2$ do not occur in the clock variables of $\widehat{P}_1$, i.e $\mathsf{clk}(\widehat{P}_1) \cap \mathsf{iclk}(\widehat{P}_2) = \emptyset$, and
2. if $\widehat{P}'$ is of the form $\widehat{P}_1 \mid \widehat{P}_2$, then the clock variables of $\widehat{P}_1$ and $\widehat{P}_2$ are disjoint, i.e. $\mathsf{clk}(\widehat{P}_1) \cap \mathsf{clk}(\widehat{P}_2) = \emptyset$. $\square$

Clearly, if $\widehat{P}$ is a safely clocked process term, then $\widehat{P}$ is contention free.
For the sake of completeness, the formal definition of a *safely clocked network* is given as follows.

Definition 4.10 (Safely clocked network) A clocked network $\widehat{N} \in \widehat{Network}_K$ is said to be *safely clocked* if each channel is associated with a distinct clock variable, i.e. if

$$\forall\, k, k' \in K \mid k \neq k' \,.\, \widehat{N}_k = (\_, h) \wedge \widehat{N}_{k'} = (\_, h') \Rightarrow h \neq h'. \qquad \square$$

It can be shown that the edge relation of a TA constructed by the method of §4.3 preserves the safety of clock allocation, i.e. if a location q is safely clocked and there are $\psi_1, \ldots, \psi_n$, $\lambda_1, \ldots, \lambda_n$ and $\mathsf{H}_1, \ldots, \mathsf{H}_n$ $(n \geq 0)$ such that $q = q_0 \xrightarrow{\psi_1,\lambda_1,\mathsf{H}_1} q_1 \cdots \xrightarrow{\psi_n,\lambda_n,\mathsf{H}_n} q_n = q'$, then q' is safely clocked. The property of static control is essential for the proof, which is a long but straightforward induction and is omitted. An obvious corollary is that if the initial location is safely clocked, then all reachable locations are contention free.

The requirement of safe clock allocation is stronger than strictly necessary to ensure that the sort of problems mentioned above are avoided. However, it is a simple property which can be checked statically, and will be enforced throughout, unless its relaxation is explicitly stated and justified.

4.2.4 Clocked *bCANDLE* systems

The definitions are extended to *bCANDLE* systems in an obvious way.

Definition 4.11 (Clocked *bCANDLE* systems) The set $\widehat{bCAN}$ of clocked *bCANDLE* systems is the set of triples $(\widehat{P}, \widehat{N}, D)$ where $\widehat{P} \in \widehat{Proc}$ is a safely clocked process term with static control, $\widehat{N}$ is a safely clocked network in $\widehat{Network}$, D is a data environment in *DataEnv*, and the following conditions are satisfied:

- the sets of process and network clocks are disjoint, i.e. $\mathsf{clk}(\widehat{P}) \cap \mathsf{clk}(\widehat{N}) = \emptyset$;
- the corresponding unclocked system $(\mathsf{unclk}(\widehat{P}), \mathsf{unclk}(\widehat{N}), D)$ is a *bCANDLE* system in $bCAN$. □

Definition 4.12 (Clock Variables: *bCANDLE* system) The clock variables of a *bCANDLE* system $\widehat{B} \in \widehat{bCAN}$ are identified by the function $\mathsf{clk} : \widehat{bCAN} \rightarrow 2^{\mathcal{H}}$, where $\mathsf{clk}(\widehat{P}, \widehat{N}, D) \mathrel{\widehat{=}} \mathsf{clk}(\widehat{P}) \cup \mathsf{clk}(\widehat{N})$. □

Definition 4.13 (Unclocked *bCANDLE* system) The unclocked *bCANDLE* system corresponding to a clocked *bCANDLE* system is given by the function $\mathsf{unclk} : \widehat{bCAN} \rightarrow bCAN$, where $\mathsf{unclk}(\widehat{P}, \widehat{N}, D) \mathrel{\widehat{=}} (\mathsf{unclk}(\widehat{P}), \mathsf{unclk}(\widehat{N}), D)$. □

4.3 Timed Automaton Construction

4.3.1 Principles of construction

The TA for a clocked *bCANDLE* system $\widehat{B} \in \widehat{bCAN}$ has some subset of $\widehat{bCAN}$ as its set of locations with $\widehat{B}$ as the initial location. The set $\mathcal{H}$ of clocks comprises the set $\mathsf{clk}(\widehat{B})$ of clocks occurring in $\widehat{B}$, together with a distinct *urgent* clock $h_u \notin \mathsf{clk}(\widehat{B})$, used in enforcing immediate actions. The set A of actions contains the sets of process and network actions $A_\mathsf{p} \cup A_\mathsf{n}$. The definition of the construction of the edges of the TA closely follows the standard semantic rules for the corresponding unclocked system (§3.6). For each rule in the semantics which justifies a transition labelled with a discrete action, there is a corresponding rule which introduces an edge in the automaton. Similarly, the rules which justify the time transitions are captured by the definition of the invariant function I. This style of presentation, adopted also in [Nic92, DB96], emphasises the relationship between the semantics of a system model and its associated TA.

4.3.2 Construction of the automaton

We begin by explaining the notion of *structurally reachable* location which is a useful auxiliary concept in the definition of the TA construction.

A location q is structurally reachable if there is a sequence of edges from the initial location $q^{\mathcal{I}}$ to q, i.e. there are $\psi_1, \ldots, \psi_n$, $\lambda_1, \ldots, \lambda_n$ and $\mathsf{H}_1, \ldots, \mathsf{H}_n$ ($n \geq 0$) such that $q^{\mathcal{I}} = q_0 \overset{\psi_1,\lambda_1,\mathsf{H}_1}{\longrightarrow} q_1 \cdots \overset{\psi_n,\lambda_n,\mathsf{H}_n}{\longrightarrow} q_n = q$. The structurally reachable part of an automaton $\mathcal{A}$ is the automaton $\mathsf{sreach}(\mathcal{A})$ which is given by the restriction to structurally reachable locations. We use the term "structural reachability" for this concept since it is based on the structure of an automaton as a directed graph, and is different from the usual concept of reachability in the transition system of the automaton (see Definitions 2.5 and 2.11).

Definition 4.14 Let $\mathcal{A} = (Q, q^{\mathcal{I}}, A, \mathcal{H}, E, I)$ be an automaton. Then, the structurally reachable part of $\mathcal{A}$ is denoted $\mathsf{sreach}(\mathcal{A})$ and is defined to be the automaton $(Q', q^{\mathcal{I}}, A, \mathcal{H}, E', I')$, where

- Q' is the least set satisfying

  1. $q^{\mathcal{I}} \in Q'$
  2. if $q \in Q'$ and $(q, \_, \_, \_, q') \in E$ then $q' \in Q'$

  and

- $E' = E \cap (Q' \times \Psi_{\mathcal{H}} \times A \times 2^{\mathcal{H}} \times Q')$

- $I' = I \cap (Q' \times \Psi_{\mathcal{H}})$ $\square$

Remark 4.2 For any TA $\mathcal{A}$, it is clear that the transition systems of $\mathcal{A}$ and $\mathsf{sreach}(\mathcal{A})$ are strongly equivalent.

Now, we can formally define the construction of a TA corresponding to a *bCANDLE* system.

Definition 4.15 (Timed automaton construction) The timed automaton for a clocked *bCANDLE* system $\widehat{B} \in b\widehat{CAN}$ is denoted $\mathcal{G}(\widehat{B})$, where $\mathcal{G}(\widehat{B}) \mathrel{\widehat{=}} \mathsf{sreach}(\mathcal{G}^+(\widehat{B}))$ and $\mathcal{G}^+(\widehat{B})$ is the automaton $(Q, q^{\mathcal{I}}, A, \mathcal{H}, E, I)$, where

- $Q = b\widehat{CAN}$ is the set of locations.

- $q^{\mathcal{I}} = \widehat{B}$ is the initial location.

$$\textbf{E\_N.1}\ \frac{\widehat{N}_k = (\Downarrow, m : u)^h}{\widehat{N} \xrightarrow{\mathsf{tt}, k \rightsquigarrow m, \{h_u, h\}}_{\mathsf{n}} \widehat{N}[k := (\overset{\mathsf{lb},\mathsf{ub}}{\rightsquigarrow} m, u)^h]}$$

$$\textbf{E\_N.2}\ \frac{\widehat{N}_k = (\overset{t_1,t_2}{\rightsquigarrow} m, u)^h \wedge t_1 \in \mathbb{N}}{\widehat{N} \xrightarrow{h \geq t_1, k \uparrow m, \{h_u\}}_{\mathsf{n}} \widehat{N}[k := (\uparrow m, u)^h]}$$

$$\textbf{E\_N.3}\ \frac{\widehat{N}_k = (\uparrow m, u)^h}{\widehat{N} \xrightarrow{\mathsf{tt}, m \rightsquigarrow k, \{h_u, h\}}_{\mathsf{n}} \widehat{N}[k := (m \overset{\mathsf{lB},\mathsf{uB}}{\rightsquigarrow}, u)^h]}$$

$$\textbf{E\_N.4}\ \frac{\widehat{N}_k = (\_ \overset{t_1,t_2}{\rightsquigarrow}, u)^h \wedge t_1 \in \mathbb{N}}{\widehat{N} \xrightarrow{h \geq t_1, k\downarrow, \{h_u\}}_{\mathsf{n}} \widehat{N}[k := (\Downarrow, u)^h]}$$

Fig. 4.3: Rules for Network Edges

- $A = A_{\mathsf{p}} \cup A_{\mathsf{n}}$ is the set of action labels.
- $\mathcal{H} = \mathsf{clk}(\widehat{B}) \cup \{h_u\}$ is the set of clock variables, where $h_u \notin \mathsf{clk}(\widehat{B})$.
- E is the least set of edges which is closed under the rules of figures 4.3, 4.4, 4.5 and 4.6. The rules make use of generic labels $\lambda_{\mathsf{p}}, \lambda_{\mathsf{n}}$ and λ, where λ_{p} ranges over A_{p}, λ_{n} ranges over A_{n} and λ ranges over A.
- $I : Q \to \Psi_{\mathcal{H}}$ is the invariant function as defined in Definition 4.16. □

Definition 4.16 (Invariant Function) Let $\mathcal{H}$ be a set of clock variables and let $\widehat{B} \in b\widehat{CAN}$ be a clocked *bCANDLE* system, where $\mathsf{clk}(\widehat{B}) \cup \{h_u\} \subseteq \mathcal{H}$. The *invariant function*, $I : b\widehat{CAN} \to \Psi_{\mathcal{H}}$ is as defined in Figure 4.7. □

4.3.3 Commentary on the construction

The translation of a *bCANDLE* system to its associated TA is straightforward for the most part. However, there are some aspects which require clarification. These concern the treatment of data and the enforcement of urgent transitions. These points are considered below.

Treatment of Data

The TA constructed by the method described here are exactly the timed safety automata (TSA) defined in [HNSY94]. These automata have been studied extensively and can be analysed automatically using tools such as KRONOS [BDM+98]. The choice of TSA as the target of the translation from *bCANDLE* directs the translation process in a number of ways. In the treatment of data, it requires that each distinct reachable data environment gives rise to at least one distinct location in the corresponding TA. Very often, this leads

$$\textbf{E\_Snd.1}\quad \frac{\widehat{N}_k = (s, u) \wedge v = D.x}{(k!i.x, \widehat{N}, D) \xrightarrow{\mathtt{tt}, k!i.v, \{h_u\}} (\checkmark, \widehat{N}[k := (s, u \looparrowleft i.v)], D)}$$

$$\textbf{E\_Snd.2}\quad \frac{\widehat{N} \xrightarrow{\psi, \lambda_{\mathsf{n}}, \mathsf{H}}_{\mathsf{n}} \widehat{N}'}{(k!i.x, \widehat{N}, D) \xrightarrow{\psi, \lambda_{\mathsf{n}}, \mathsf{H}} (k!i.x, \widehat{N}', D)}$$

$$\textbf{E\_Rcv.1}\quad \frac{\widehat{N}_k = (\uparrow i.v, \_)}{(k?i.x, \widehat{N}, D) \xrightarrow{\mathtt{tt}, k?i.v, \{h_u\}} (\checkmark, \widehat{N}, D[x := v])}$$

$$\textbf{E\_Rcv.2}\quad \frac{\widehat{N} \xrightarrow{\psi, \lambda_{\mathsf{n}}, \mathsf{H}}_{\mathsf{n}} \widehat{N}' \wedge (\widehat{N}_k \neq (\uparrow i.\_, \_) \vee \widehat{N}_k = \widehat{N}'_k)}{(k?i.x, \widehat{N}, D) \xrightarrow{\psi, \lambda_{\mathsf{n}}, \mathsf{H}} (k?i.x, \widehat{N}', D)}$$

$$\textbf{E\_Comp.1}\quad \frac{D \xrightarrow{\omega}_{\mathsf{d}} D' \wedge t_1 \in \mathbb{N}}{([\omega : t_1, t_2]^h, \widehat{N}, D) \xrightarrow{h \geq t_1, \omega, \{h_u\}} (\checkmark, \widehat{N}, D')}$$

$$\textbf{E\_Comp.2}\quad \frac{\widehat{N} \xrightarrow{\psi, \lambda_{\mathsf{n}}, \mathsf{H}}_{\mathsf{n}} \widehat{N}'}{([\omega : t_1, t_2]^h, \widehat{N}, D) \xrightarrow{\psi, \lambda_{\mathsf{n}}, \mathsf{H}} ([\omega : t_1, t_2]^h, \widehat{N}', D)}$$

Fig. 4.4: Rules for Basic System Edges

to the creation of a TA with a large number of locations. Other approaches which accommodate explicit data in TA models have avoided this problem by working with extended automata; the usual extension being to allow the use of conditions over data variables on edges, in addition to conditions over clock variables, see for example [Tri98, Her98, BLL$^+$98, BLSTV99]. A single location may then represent many control states, each having a different data environment. If the user is expected to create a system model explicitly as a network of TA, then the handling of data is done most sensibly using extended automata of this sort. Certainly, one would not wish to construct by hand the automata which are created by our method. However the situation is not so clear when creating automata automatically from some other input language, as is the case here for *bCANDLE*. For although the construction may give rise to many more locations in the TA than a construction for extended automata, it does not lead to an increase in the number of states in the simulation graph (§2.7.6), which is the primary structure over which most analyses are performed and whose size is their main constraining factor. This point will be considered further in Chapter 5.

Urgent transitions

There are several operations in *bCANDLE* which must either be executed or disabled *without delay*, such operations are called *urgent*. The urgent operations of the process component of a *bCANDLE* model are

- all send operations, $k!i.x$,

$$\textbf{E\_Gu.1}\ \frac{D \models \gamma}{(\gamma \to \widehat{P}, \widehat{N}, D) \xrightarrow{\mathtt{tt}, \gamma, \{h_u\} \cup \mathsf{iclk}(\widehat{P})} (\widehat{P}, \widehat{N}, D)}$$

$$\textbf{E\_Gu.2}\ \frac{\widehat{N} \xrightarrow{\psi, \lambda_\mathsf{n}, \mathsf{H}}_{\mathsf{n}} \widehat{N}'}{(\gamma \to \widehat{P}, \widehat{N}, D) \xrightarrow{\psi, \lambda_\mathsf{n}, \mathsf{H}} (\gamma \to \widehat{P}, \widehat{N}', D)}$$

$$\textbf{E\_Seq.1}\ \frac{(\widehat{P}, \widehat{N}, D) \xrightarrow{\psi, \lambda, \mathsf{H}} (\widehat{P}', \widehat{N}', D') \wedge \widehat{P}' \not\equiv \checkmark}{(\widehat{P}\ ;\ \widehat{Q}, \widehat{N}, D) \xrightarrow{\psi, \lambda, \mathsf{H}} (\widehat{P}'\ ;\ \widehat{Q}, \widehat{N}', D')}$$

$$\textbf{E\_Seq.2}\ \frac{(\widehat{P}, \widehat{N}, D) \xrightarrow{\psi, \lambda_\mathsf{p}, \mathsf{H}} (\checkmark, \widehat{N}', D')}{(\widehat{P}\ ;\ \widehat{Q}, \widehat{N}, D) \xrightarrow{\psi, \lambda_\mathsf{p}, \mathsf{H} \cup \mathsf{iclk}(\widehat{Q})} (\widehat{Q}, \widehat{N}', D')}$$

$$\textbf{E\_Ch.1}\ \frac{(\widehat{P}, \widehat{N}, D) \xrightarrow{\psi, \lambda_\mathsf{p}, \mathsf{H}} (\widehat{P}', \widehat{N}', D')}{(\widehat{P} + \widehat{Q}, \widehat{N}, D) \xrightarrow{\psi, \lambda_\mathsf{p}, \mathsf{H}} (\widehat{P}', \widehat{N}', D')}$$

$$\textbf{E\_Ch.2}\ \frac{(\widehat{Q}, \widehat{N}, D) \xrightarrow{\psi, \lambda_\mathsf{p}, \mathsf{H}} (\widehat{Q}', \widehat{N}', D')}{(\widehat{P} + \widehat{Q}, \widehat{N}, D) \xrightarrow{\psi, \lambda_\mathsf{p}, \mathsf{H}} (\widehat{Q}', \widehat{N}', D')}$$

$$\textbf{E\_Ch.3}\ \frac{(\widehat{P}, \widehat{N}, D) \xrightarrow{\psi, \lambda_\mathsf{n}, \mathsf{H}} (\widehat{P}, \widehat{N}', D) \wedge (\widehat{Q}, \widehat{N}, D) \xrightarrow{\psi, \lambda_\mathsf{n}, \mathsf{H}} (\widehat{Q}, \widehat{N}', D)}{(\widehat{P} + \widehat{Q}, \widehat{N}, D) \xrightarrow{\psi, \lambda_\mathsf{n}, \mathsf{H}} (\widehat{P} + \widehat{Q}, \widehat{N}', D)}$$

$$\textbf{E\_Rec}\ \frac{(\widehat{P}[\mathrm{rec}\, X.\widehat{P}/X], \widehat{N}, D) \xrightarrow{\psi, \lambda, \mathsf{H}} (\widehat{P}', \widehat{N}', D')}{(\mathrm{rec}\, X.\widehat{P}, \widehat{N}, D) \xrightarrow{\psi, \lambda, \mathsf{H}} (\widehat{P}', \widehat{N}', D')}$$

Fig. 4.5: Rules for Guard, Sequential Composition, Choice and Recursion Edges

- data guarded operations, $\gamma \to P$, when the guard γ is satisfied, and
- computations for which the upper bound is 0, i.e. computations of the form $[\omega : 0, 0]$.

The urgent network operations are

- the commencement of the transmission of the highest priority pending message when a channel is free and its message queue is not empty, i.e. transitions of the form $k \rightsquigarrow m$, and
- the commencement of the post-acceptance phase of message transmission, i.e. transitions of the form $m \rightsquigarrow k$.

The urgency of these operations is enforced in a TA by using a single distinct clock variable h_u, which is reset on every edge and which is used in the invariant $h_u \leq 0$, attached to all locations in which an urgent transition is enabled. Notice that the clock guard on all urgent transitions is $\mathtt{tt}$.

$$\textbf{E\_Int.1}\ \frac{(\widehat{P},\widehat{N},D)\xrightarrow{\psi,\lambda_{\mathsf{p}},\mathsf{H}}(\widehat{P}',\widehat{N}',D')\wedge\widehat{P}'\not\equiv\checkmark}{(\widehat{P}\,[>\widehat{Q},\widehat{N},D)\xrightarrow{\psi,\lambda_{\mathsf{p}},\mathsf{H}}(\widehat{P}'\,[>\widehat{Q},\widehat{N}',D')}$$

$$\textbf{E\_Int.2}\ \frac{(\widehat{P},\widehat{N},D)\xrightarrow{\psi,\lambda_{\mathsf{p}},\mathsf{H}}(\checkmark,\widehat{N}',D')}{(\widehat{P}\,[>\widehat{Q},\widehat{N},D)\xrightarrow{\psi,\lambda_{\mathsf{p}},\mathsf{H}}(\checkmark,\widehat{N}',D')}$$

$$\textbf{E\_Int.3}\ \frac{(\widehat{Q},\widehat{N},D)\xrightarrow{\psi,\lambda_{\mathsf{p}},\mathsf{H}}(\widehat{Q}',\widehat{N}',D')}{(\widehat{P}\,[>\widehat{Q},\widehat{N},D)\xrightarrow{\psi,\lambda_{\mathsf{p}},\mathsf{H}}(\widehat{Q}',\widehat{N}',D')}$$

$$\textbf{E\_Int.4}\ \frac{(\widehat{P},\widehat{N},D)\xrightarrow{\psi,\lambda_{\mathsf{n}},\mathsf{H}}(\widehat{P},\widehat{N}',D)\wedge(\widehat{Q},\widehat{N},D)\xrightarrow{\psi,\lambda_{\mathsf{n}},\mathsf{H}}(\widehat{Q},\widehat{N}',D)}{(\widehat{P}\,[>\widehat{Q},\widehat{N},D)\xrightarrow{\psi,\lambda_{\mathsf{n}},\mathsf{H}}(\widehat{P}\,[>\widehat{Q},\widehat{N}',D)}$$

$$\textbf{E\_Par.1}\ \frac{(\widehat{P},\widehat{N},D)\xrightarrow{\psi,\lambda_{\mathsf{p}},\mathsf{H}}(\widehat{P}',\widehat{N}',D')\wedge\widehat{P}'\not\equiv\checkmark}{(\widehat{P}\mid\widehat{Q},\widehat{N},D)\xrightarrow{\psi,\lambda_{\mathsf{p}},\mathsf{H}}(\widehat{P}'\mid\widehat{Q},\widehat{N}',D')}$$

$$\textbf{E\_Par.2}\ \frac{(\widehat{P},\widehat{N},D)\xrightarrow{\psi,\lambda_{\mathsf{p}},\mathsf{H}}(\checkmark,\widehat{N}',D')}{(\widehat{P}\mid\widehat{Q},\widehat{N},D)\xrightarrow{\psi,\lambda_{\mathsf{p}},\mathsf{H}}(\widehat{Q},\widehat{N}',D')}$$

$$\textbf{E\_Par.3}\ \frac{(\widehat{Q},\widehat{N},D)\xrightarrow{\psi,\lambda_{\mathsf{p}},\mathsf{H}}(\widehat{Q}',\widehat{N}',D')\wedge\widehat{Q}'\not\equiv\checkmark}{(\widehat{P}\mid\widehat{Q},\widehat{N},D)\xrightarrow{\psi,\lambda_{\mathsf{p}},\mathsf{H}}(\widehat{P}\mid\widehat{Q}',\widehat{N}',D')}$$

$$\textbf{E\_Par.4}\ \frac{(\widehat{Q},\widehat{N},D)\xrightarrow{\psi,\lambda_{\mathsf{p}},\mathsf{H}}(\checkmark,\widehat{N}',D')}{(\widehat{P}\mid\widehat{Q},\widehat{N},D)\xrightarrow{\psi,\lambda_{\mathsf{p}},\mathsf{H}}(\widehat{P},\widehat{N}',D')}$$

$$\textbf{E\_Par.5}\ \frac{(\widehat{P},\widehat{N},D)\xrightarrow{\psi,\lambda_{\mathsf{n}},\mathsf{H}}(\widehat{P},\widehat{N}',D)\wedge(\widehat{Q},\widehat{N},D)\xrightarrow{\psi,\lambda_{\mathsf{n}},\mathsf{H}}(\widehat{Q},\widehat{N}',D)}{(\widehat{P}\mid\widehat{Q},\widehat{N},D)\xrightarrow{\psi,\lambda_{\mathsf{n}},\mathsf{H}}(\widehat{P}\mid\widehat{Q},\widehat{N}',D)}$$

Fig. 4.6: Rules for Interrupt and Parallel Composition Edges

Proposition 4.1 *Let $\widehat{B} \in \widehat{bCAN}$ be a clocked bCANDLE system and $\mathcal{G}(\widehat{B}) = (Q, q^{\mathcal{I}}, A, \mathcal{H}, E, I)$ the TA constructed from $\widehat{B}$ according to Definition 4.15. Then, for any edge $e = (q, \psi, \lambda, \mathsf{H}, q') \in E$, the urgent clock is reset by e, i.e. $h_u \in \mathsf{H}$.*

Proof Induction on the depth of the inference justifying the existence of the edge. Intuitively, one can see that h_u is reset by every basic process and network edge and that the resets are propagated by all process operators. □

4.3.4 Correctness of the construction

The TA generated from a *bCANDLE* system model yields a transition system which is strongly equivalent to that given by the standard *bCANDLE* semantics. Therefore, we can be confident that conclusions reached by analysing the TA

$$
\begin{array}{rcl}
I(\widehat{P}, \widehat{N}, D) & \widehat{=} & I(\widehat{P}, D) \wedge I(\widehat{N}) \\
I(k!i.x, D) & \widehat{=} & h_u \leq 0 \\
I(k?i.x, D) & \widehat{=} & \mathsf{tt} \\
I([\omega : t_1, t_2]^h, D) & \widehat{=} & \mathsf{if}\ t_2 \in \mathbb{N}\ \mathsf{then}\ h \leq t_2\ \mathsf{else}\ \mathsf{tt} \\
I(\gamma \to \widehat{P}, D) & \widehat{=} & \mathsf{if}\ D \models \gamma\ \mathsf{then}\ h_u \leq 0\ \mathsf{else}\ \mathsf{tt} \\
I(\widehat{P}_1 \,;\, \widehat{P}_2, D) & \widehat{=} & I(\widehat{P}_1, D) \\
I(\widehat{P}_1 \bowtie \widehat{P}_2, D) & \widehat{=} & I(\widehat{P}_1, D) \wedge I(\widehat{P}_2, D) \qquad \bowtie \in \{+, [>, |\} \\
I(\mathrm{rec}\, X.\widehat{P}, D) & \widehat{=} & I(\widehat{P}[\mathrm{rec}\, X.\widehat{P}/X], D) \\
I(\widehat{N}) & \widehat{=} & \displaystyle\bigwedge_{k \in K} I(\widehat{N}_k) \\
I(\Downarrow, \langle\rangle)^h & \widehat{=} & \mathsf{tt} \\
I(\Downarrow, m : u)^h & \widehat{=} & h_u \leq 0 \\
I(\overset{t_1,t_2}{\rightsquigarrow} m, u)^h & \widehat{=} & \mathsf{if}\ t_2 \in \mathbb{N}\ \mathsf{then}\ h \leq t_2\ \mathsf{else}\ \mathsf{tt} \\
I(\uparrow m, u)^h & \widehat{=} & h_u \leq 0 \\
I(m \overset{t_1,t_2}{\rightsquigarrow}, u)^h & \widehat{=} & \mathsf{if}\ t_2 \in \mathbb{N}\ \mathsf{then}\ h \leq t_2\ \mathsf{else}\ \mathsf{tt}
\end{array}
$$

Fig. 4.7: Invariant function $I : b\widehat{CAN} \to \Psi_{\mathcal{H}}$

are valid for the system model. We state the equivalence formally below but relegate the details of the proof to Appendix B.

Proposition 4.2 *Let $\widehat{B} \in b\widehat{CAN}$ be a clocked bCANDLE system and $B \,\widehat{=}\, \mathsf{unclk}(\widehat{B})$ the corresponding unclocked system. Let $\mathcal{G}(\widehat{B})$ be the TA given by Definition 4.15. Then, the transition systems of $\mathcal{G}(\widehat{B})$ and B are strongly equivalent.*

$$\mathcal{T}\,[\![\,\mathcal{G}(\widehat{B})]\!] \leftrightarroweq \mathcal{T}\,[\![B]\!]$$

Proof Appendix B. □

4.4 Implementation of the construction

Although §4.3.2 gives a precise description of the TA which corresponds to a *bCANDLE* system, it does not give a practical method for constructing it.

A significant difficulty, in practice, concerns the size of the representation of locations $(\widehat{P}, \widehat{N}, D) \in b\widehat{CAN}$. In particular, the representation of the control component $\widehat{P}$ by an algebraic term results in implementations which are extremely inefficient in their use of computer memory. Moreover, a construction of the TA based upon repeated construction and/or comparison of the TA of the sub-systems, is not time-efficient.

Similar problems have been observed by Garavel in the translation of LOTOS [Gar92], and by Yovine in the translation of ATP [Yov93]. We adapt their solutions to our system models, in developing an approach which accommodates both explicit data values and dense real-time. In this approach, the translation of a system model into a TA is performed in two stages:

- in the first stage, a compact, intermediate form, similar to a Petri net [Mur89], is constructed for the system model;
- in the second stage, the TA itself is constructed efficiently using the net built in the first stage.

The main advantage of using a net as an intermediate representation is that it is then possible to represent the control component of a system state by a marking of the net. This representation is likely to be much more compact than the abstract syntax tree of the corresponding process term, and it leads to algorithms with a reduced need to manipulate sub-terms.

The remainder of this section is concerned with the development of an efficient method for constructing the TA of a *bCANDLE* system $\widehat{B}$, i.e. with the construction of $\mathcal{G}(\widehat{B})$.

4.4.1 Nets

Introduction

The nets which are used in this work are not strictly Petri nets but are close to the extended nets of [Yov93]. As usual, a net consists of a set of *places* and a set of *transitions*; the convention used here is to denote a set of places by W, W', W_1 etc. and a set of transitions by $\Theta, \Theta', \Theta_1$ etc. Two main extensions are introduced.

1. Each transition has an associated *attribute* which is used in determining whether or not the transition is *fireable* in a given system *context*, where a context consists of a network and a data environment. This is in accord with many of the varieties of generalised or interpreted Petri net [Kel76, Sif77].
2. In addition to a source set of places which must be *marked* in order for a transition to be fireable, and a target set of places to which control flows when a transition fires, each transition is also associated with a set of places which are said to be *vulnerable* to the firing of the transition. When a transition fires, control is removed not only from the places in its source set but also from all those places which are vulnerable to it. This extension allows a compact representation of the interrupt operator in particular.

Example 4.3 Figure 4.8 shows an example net. It represents the process term $k?\mathit{flow}.y\,;\,[\mathit{AdjustValve} : 200, 300]^{H1}\,[> [450, 500]^{H2}\,;\,\text{idle}$. The places of the net are shown as circles and the transitions as boxes. The shaded circles represent

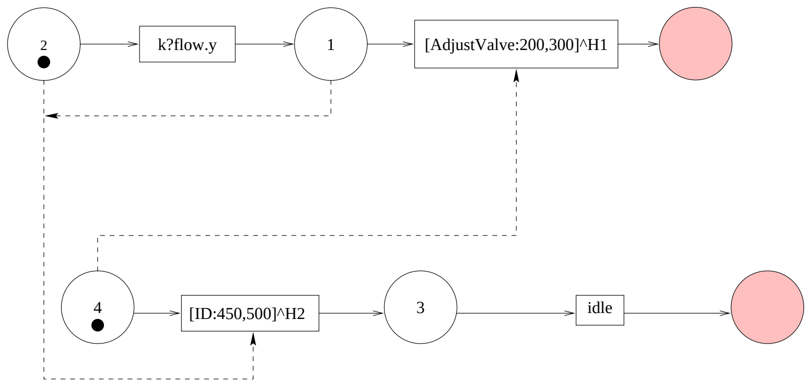


Fig. 4.8: Example Net

a distinguished place tick[1], modelling termination. A label inside a transition box denotes the transition attribute, e.g., $k?flow.y$. The standard flow relation is shown using solid lines, e.g., if place 2 is marked, and the context allows, the transition $k?flow.y$ can fire, removing a token from place 2 and adding a token to place 1. The vulnerability relation is shown using dashed lines, e.g., places 1 and 2 are vulnerable to the firing of the transition $[ID : 450, 500]^{H2}$, so a token in either of those places is removed when the transition fires. The small black circles in places 2 and 4 show that those places are marked. $\square$

Nets are introduced formally below.

Definitions and Notation

Let $\widehat{Proc}$ be a set of clocked process terms over process variables $\mathcal{X}$, predicate names Γ and clocks $\mathcal{H}$. The set *Attribute* of *transition attributes* is defined:

$$\alpha ::= \widehat{\beta} \mid \langle\gamma\rangle \mid X$$

where $\alpha \in Attribute$ is a transition attribute, $\widehat{\beta} \in \widehat{Proc}$ is a clocked basic term and $X \in \mathcal{X}$ is a process variable. We use the notation $\langle\gamma\rangle$ to denote a transition attribute consisting of the predicate name $\gamma \in \Gamma$.

The set of clocks associated with a transition attribute α is denoted $\mathsf{clk}(\alpha)$, where $\mathsf{clk}([\omega : t_1, t_2]^h) \mathrel{\widehat{=}} \{h\}$, and $\mathsf{clk}(\alpha) \mathrel{\widehat{=}} \emptyset$, for any attribute α of the form $k!i.x$, $k?i.x$, $\langle\gamma\rangle$, and X.

A net can now be defined as follows.

Definition 4.17 (Net) A net is a tuple $\mathcal{R} = (W, \Theta, W^I)$ where

- W is the set of *places*

[1] In the diagram of a net, we often have more than one shaded circle, in order to simplify the layout. All such shaded circles should be interpreted as representing the same tick place.

- $\Theta \subseteq W \times 2^W \times \mathit{Attribute} \times 2^{W^{\checkmark}}$ is the set of *transitions*. $W^{\checkmark}$ denotes the set of places $W \cup \{\mathsf{tick}\}$ in which $\mathsf{tick} \notin W$ is a distinguished place used in the representation of the terminal process $\checkmark$.

- $W^I \subseteq W$ is the set of *initial* places □

Let $\mathcal{R} = (W, \Theta, W^I)$ be a net and $\theta = (w, W^V, \alpha, W^T) \in \Theta$ a transition. We adopt the following conventions:

- $w \in W$ is the *trigger* of θ, denoted ${}^{\bullet}\theta$.

- $W^V \subseteq W$ is the set of places *vulnerable to* θ, denoted ${}^{\circ}\theta$

- $\alpha \in \mathit{Attribute}$ is the *attribute* of θ, denoted $\alpha\theta$

- $W^T \subseteq W$ is the *target* set of θ, denoted $\theta^{\bullet}$

In the case that a place w is the trigger of exactly one transition, the transition triggered by w is denoted by θ_w. Every place in a net constructed from a *bCANDLE* system according to the method of §4.4.2 is the trigger of exactly one transition.

A *marking* is a set of places. The marking W^I is the *initial marking*. Let W be a marking. For each transition θ, if ${}^{\bullet}\theta \in W$, then θ is said to be *conditionally enabled* in W.

Let $\mathcal{R}_i = (W_i, \Theta_i, W_i^I)$ for $i \in \{1, 2\}$, be two nets. $\mathcal{R}_1$ and $\mathcal{R}_2$ are said to be *disjoint* iff $W_1 \cap W_2 = \emptyset$.

The set of clocks associated with a set W of places is denoted $\mathsf{clk}(W)$, where $\mathsf{clk}(W) \mathrel{\widehat{=}} \bigcup_{w \in W} \mathsf{clk}(\alpha\theta_w)$.

Behaviour

The semantics of a net is given with respect to a system context which comprises a network and a data environment. The semantics is given as a transition system between states consisting of a marking of the net and a context. Given a net $\mathcal{R} = (W, \Theta, W^I)$, a state $(\widehat{P}, \widehat{N}, D) \in \widehat{bCAN}$ can be represented by $(W_1, \widehat{N}, D)$ where $W_1 \subseteq W$ is a marking of $\mathcal{R}$ which represents the control component $\widehat{P}$. Intuitively, a system can evolve from one state $(W_1, \widehat{N}, D)$ to another state $(W_2, \widehat{N}', D')$ as the result of either a process transition or a network transition.

For a process transition, assume $w \in W_1$ and that w is the trigger of some transition θ. If the context $\widehat{N}, D$ satisfies the conditions required by the attribute $\alpha\theta$, then a new marking W_2 is created from W_1 by removing w and any places which are vulnerable to θ, and then including all of the target places of θ. The new context, $\widehat{N}', D'$ is created according to the requirements of the attribute $\alpha\theta$.

In the case of a network transition, the system may evolve to a new state, in which the network component is modified, but the marking and data environment remain unchanged. However, notice that a network transition is inhibited by a marking as follows:

- a message offer cannot be removed if some process is ready to accept it, i.e., a network transition to the post-acceptance phase of transmission of a message with identifier i on a channel k is not allowed if the current marking contains a place which is the trigger of a transition whose attribute is $k?i.x$ for some data variable x.

These ideas are presented formally in the rules **R.1** and **R.2** below.

Definition 4.18 Let $\mathcal{R} = (W, \Theta, W^I)$ be a net, $W_1 \subseteq W$. Let $\widehat{N}$ be a clocked network over sets K of channel identifiers and I of message identifiers. Let D be a data environment.

The process transitions of $(W_1, \widehat{N}, D)$ are given by the rule:

$$\textbf{R.1}\ \frac{\begin{array}{c} w \in W_1 \wedge (w, W^V, \alpha, W^T) \in \Theta \wedge \mathsf{fire}(\alpha, \widehat{N}, D, \psi, \lambda, \mathsf{H}', \widehat{N}', D') \wedge \\ W_2 = W_1 \setminus (\{w\} \cup W^V) \cup W^T \wedge \mathsf{H} = \mathsf{H}' \cup \mathsf{clk}(W^T) \end{array}}{(W_1, \widehat{N}, D) \xrightarrow{\psi, \lambda, \mathsf{H}}_{\mathcal{R}} (W_2, \widehat{N}', D')}$$

and the network transitions by the rule:

$$\textbf{R.2}\ \frac{\begin{array}{c} \widehat{N} \xrightarrow{\psi, \lambda_{\mathsf{n}}, \mathsf{H}}_{\mathsf{n}} \widehat{N}' \wedge \\ \forall\, k \in K, i \in I\ .\ (\neg\, \mathsf{awaited}(W, k, i) \vee \widehat{N}_k \neq (\uparrow i.\_, \_) \vee \widehat{N}_k = \widehat{N}'_k) \end{array}}{(W, \widehat{N}, D) \xrightarrow{\psi, \lambda_{\mathsf{n}}, \mathsf{H}}_{\mathcal{R}} (W, \widehat{N}', D)}$$

where

- the fire relation, as given in Figure 4.9, simply recasts the semantic rules for basic terms and guards, in defining the behaviour of each transition attribute in a given system context, and
- $\mathsf{awaited}(W, k, i)$ holds iff, in the marking W, it is possible to receive from channel k a message with identifier i. Formally,
$$\mathsf{awaited}(W, k, i) \mathrel{\widehat{=}} \{w \in W \mid \alpha\theta_w = k?i.\_\} \neq \emptyset \qquad \square$$

4.4.2 Constructing the net for a clocked term

Let $(\widehat{P}, \widehat{N}, D) \in b\widehat{CAN}$ be a *bCANDLE* system. In this section, it is shown how to construct the net for $\widehat{P}$, denoted $\mathcal{N}[\![\widehat{P}]\!]$. We begin by considering the construction of nets for the basic terms. The construction of a net for a compound term, $\widehat{P}_1 \,;\, \widehat{P}_2$, $\widehat{P}_1 + \widehat{P}_2$, $\widehat{P}_1 \,[>\, \widehat{P}_2$ and $\widehat{P}_1 \mid \widehat{P}_2$, proceeds compositionally from the nets for $\widehat{P}_1$ and $\widehat{P}_2$.

Basic Terms

The net for each of the clocked basic terms, $\widehat{\beta} \in \widehat{Proc}$ is constructed in the same way for each. A new place is created to act as the trigger of a transition whose attribute is the term itself and whose outgoing arc leads to the distinguished place, tick.

$$\textbf{F\_Snd}\ \frac{\widehat{N}_k = (s, u) \wedge v = D.x}{\mathsf{fire}(k!i.x, \widehat{N}, D, \mathtt{tt}, k!i.v, \{h_u\}, \widehat{N}[k := (s, u \hookleftarrow i.v)], D)}$$

$$\textbf{F\_Rcv}\ \frac{\widehat{N}_k = (\uparrow i.v, \_)}{\mathsf{fire}(k?i.x, \widehat{N}, D, \mathtt{tt}, k?i.v, \{h_u\}, \widehat{N}, D[x := v])}$$

$$\textbf{F\_Comp}\ \frac{D \xrightarrow{\omega}_{\mathsf{d}} D' \wedge t_1 \in \mathbb{N}}{\mathsf{fire}([\omega : t_1, t_2]^h, \widehat{N}, D, h \geq t_1, \omega, \{h_u\}, \widehat{N}, D')}$$

$$\textbf{F\_Gu}\ \frac{D \models \gamma}{\mathsf{fire}(\langle\gamma\rangle, \widehat{N}, D, \mathtt{tt}, \gamma, \{h_u\}, \widehat{N}, D)}$$

Fig. 4.9: Rules for fire

Example 4.4 Let $\widehat{\beta}$ be the term $k?flow.y$. Figure 4.10 shows the net given by $\mathcal{N}[\![\widehat{\beta}]\!]$. The figure shows the initial marking of the net. The terminal place tick

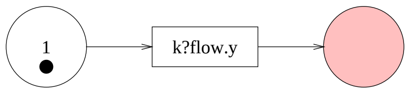


Fig. 4.10: Net for a basic term

is shown as a shaded circle. Place (1) is the trigger of the net's only transition, whose attribute is shown inside the box, and whose target set is the singleton $\{\mathsf{tick}\}$. □

Definition 4.19 Let $\widehat{\beta}$ be one of the basic terms $k!i.x$, $k?i.x$ or $[\omega : t_1, t_2]^h$. Then the net for $\widehat{\beta}$, is constructed as follows:

$$\mathcal{N}[\![\widehat{\beta}]\!] \mathrel{\widehat{=}} (\{w\}, \{(w, \{\}, \widehat{\beta}, \{\mathsf{tick}\})\}, \{w\})$$

where $w \neq \mathsf{tick}$ is a place. □

Sequential Composition

In constructing the net of the sequential composition, $\widehat{P}_1; \widehat{P}_2$, all that needs to be done is to combine the nets of $\widehat{P}_1$ and $\widehat{P}_2$ and then modify each transition θ of $\widehat{P}_1$, which leads to the immediate termination of $\widehat{P}_1$, so that it leads instead to the initial places of $\widehat{P}_2$. This represents the transfer of control from a termination point in $\widehat{P}_1$ to the starting point(s) of $\widehat{P}_2$. A transition θ leads to immediate termination iff its target set, $\theta^\bullet$, is $\{\mathsf{tick}\}$. The transfer of control is implemented simply by making $\theta^\bullet$ equal to the initial places of $\widehat{P}_2$.

Example 4.5 Consider the term $k?\mathit{flow.y}\ ;\ [\mathit{AdjustValve} : 200, 300]^{H1}$. Its net is constructed very simply, as shown in Figure 4.11. $\square$

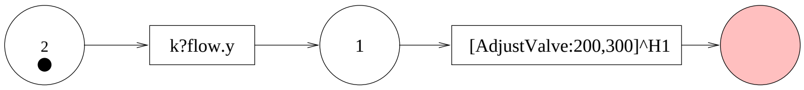


Fig. 4.11: Net for a sequential composition

Definition 4.20 Let $(W_i, \Theta_i, W_i^I) = \mathcal{N}[\![\widehat{P}_i]\!]$, for $i \in \{1, 2\}$, be disjoint nets. The net $\mathcal{N}[\![\widehat{P}_1\ ;\ \widehat{P}_2]\!]$ for the sequential composition $\widehat{P}_1\ ;\ \widehat{P}_2$ is given by

$$\mathcal{N}[\![\widehat{P}_1\ ;\ \widehat{P}_2]\!] \mathrel{\widehat{=}} (W_1 \cup W_2, \Theta_1' \cup \Theta_2, W_1^I)$$

where

$$\begin{array}{rcl} \Theta_1' & \mathrel{\widehat{=}} & \{\theta \mid \theta \in \Theta_1 \wedge \theta^\bullet \neq \{\mathsf{tick}\}\} \\ & \cup & \{({}^\bullet\theta, {}^\circ\theta, \alpha\theta, W_2^I) \mid \theta \in \Theta_1 \wedge \theta^\bullet = \{\mathsf{tick}\}\} \end{array}$$

$\square$

Guard

A guarded process $\gamma \rightarrow \widehat{P}$ evaluates the guard γ in its current data environment and then behaves as $\widehat{P}$ if the guard is true or simply idles otherwise. We construct a net rather as if γ is a basic term and $\rightarrow$ is sequential composition.

Example 4.6 Let $\widehat{P}$ be the term $\mathit{Shutdown} \rightarrow \mathsf{idle}$. Figure 4.12 shows the net given by $\mathcal{N}[\![\widehat{P}]\!]$. $\square$

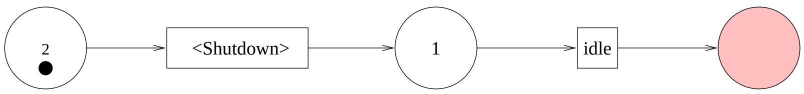


Fig. 4.12: Net for a data-guarded term

Definition 4.21 Let $\mathcal{N}[\![\widehat{P}]\!] = (W, \Theta, W^I)$, then the net of $\gamma \rightarrow \widehat{P}$ is given by

$$\mathcal{N}[\![\gamma \rightarrow \widehat{P}]\!] \mathrel{\widehat{=}} (W \cup \{w\}, \Theta \cup \{(w, \{\}, \langle\gamma\rangle, W^I)\}, \{w\})$$

where $w \notin W^{\checkmark}$ is a place. $\square$

Choice

A choice, $\widehat{P}_1 + \widehat{P}_2$, is resolved in favour of the process which is first able to perform an initial transition, the possibility of action then being removed from the other process. The removal of control is represented in the net for $\widehat{P}_1 + \widehat{P}_2$ by adjusting the vulnerable sets of the transitions of each process, so that control is removed from one process whenever an action occurs in the other.

Example 4.7
Let $\widehat{P}_1 = k?\mathit{flow}.y$; $[\mathit{AdjustValve} : 200, 300]^{H1}$ and $\widehat{P}_2 = [450, 500]^{H2}$; idle. The term $\widehat{P}_1 + \widehat{P}_2$ models the situation in which one of two possible behaviours can occur: either a *flow* message is received on channel k within 500 time units and then the process adjusts a valve; or a *flow* message is not received for at least 450 time units, after which the timeout may elapse and the process idles forever. Figure 4.13 shows the net which is constructed for $\mathcal{N}[\![\widehat{P}_1 + \widehat{P}_2]\!]$. Notice

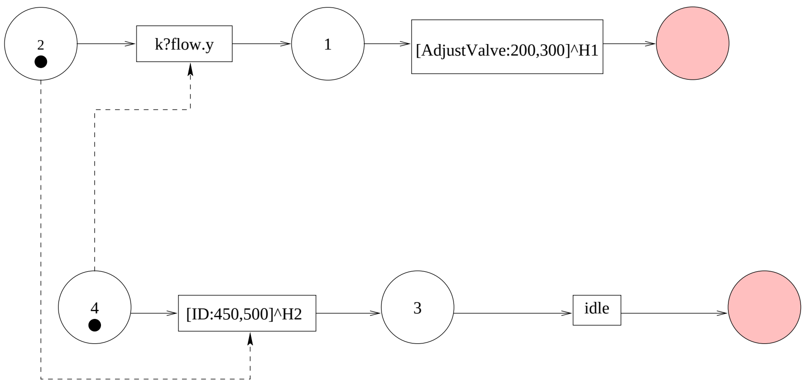


Fig. 4.13: Net for a choice

that the marking of the net shows the initial possibility of both behaviours. The places which are *vulnerable to* a transition are indicated with a dashed line, directed from each vulnerable place to the transition(s) to which it is vulnerable. For example, place 4 is vulnerable to the transition $k?\mathit{flow}.y$. This means that if $k?\mathit{flow}.y$ fires, then a token residing at place 4 will be removed, and so the transition which is triggered by it will be disabled. □

Definition 4.22 Let $(W_i, \Theta_i, W_i^I) = \mathcal{N}[\![\widehat{P}_i]\!]$, for $i \in \{1, 2\}$, be disjoint nets. Then

$$\mathcal{N}[\![\widehat{P}_1 + \widehat{P}_2]\!] \mathrel{\widehat{=}} (W_1 \cup W_2, \Theta, W_1^I \cup W_2^I)$$

where

$$\begin{array}{rl} \Theta \mathrel{\widehat{=}} & \{\theta \mid \theta \in \Theta_1 \wedge {}^\bullet\theta \notin W_1^I\} \\ \cup & \{({}^\bullet\theta, {}^\circ\theta \cup W_2^I, \alpha\theta, \theta^\bullet) \mid \theta \in \Theta_1 \wedge {}^\bullet\theta \in W_1^I\} \\ \cup & \{\theta \mid \theta \in \Theta_2 \wedge {}^\bullet\theta \notin W_2^I\} \\ \cup & \{({}^\bullet\theta, {}^\circ\theta \cup W_1^I, \alpha\theta, \theta^\bullet) \mid \theta \in \Theta_2 \wedge {}^\bullet\theta \in W_2^I\} \end{array}$$
□

Interrupt

An interrupt, $\widehat{P}_1 \,[> \widehat{P}_2$, differs from choice in that control is only removed from $\widehat{P}_2$ when a *terminating* transition of $\widehat{P}_1$ occurs. So $\widehat{P}_1$ can perform transitions while $\widehat{P}_2$ retains the possibility of action. Wherever control resides in $\widehat{P}_1$, it is removed upon the occurrence of an initial transition of $\widehat{P}_2$.

Example 4.8
Let $\widehat{P}_1 = k?\mathit{flow}.y$; $[\mathit{AdjustValve} : 200, 300]^{H1}$ and $\widehat{P}_2 = [450, 500]^{H2}$; idle. The term $\widehat{P}_1 \,[> \widehat{P}_2$ behaves similarly to $\widehat{P}_1 + \widehat{P}_2$ which was considered in Example 4.7; the primary difference being that $[450, 500]^{H2}$ acts as a watchdog timer rather than a timeout: i.e., it remains active throughout the behaviour of $\widehat{P}_1$ and is only disabled when $\widehat{P}_1$ terminates. This difference is reflected in the construction of the net for $\widehat{P}_1 \,[> \widehat{P}_2$ shown in Figure 4.14. Attention should

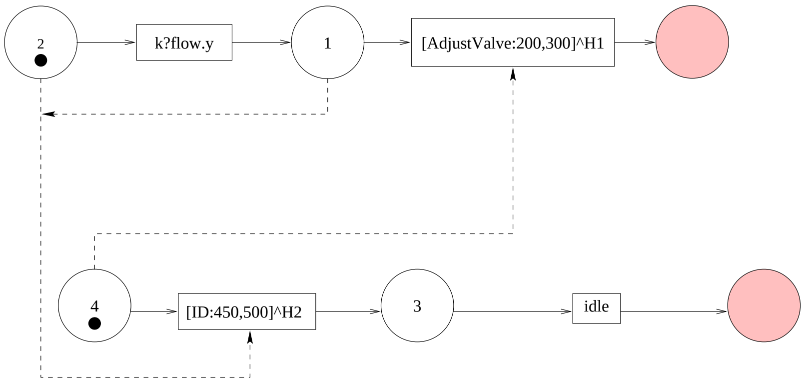


Fig. 4.14: Net for an interrupt

be given to the following points:

- the triggers of *all* transitions associated with $\widehat{P}_1$ are made vulnerable to the initial transition of $\widehat{P}_2$, and
- the trigger of the initial transition of $\widehat{P}_2$ is vulnerable only to the *terminating* transition of $\widehat{P}_1$.

The effect of this is that $[ID : 450, 500]^{H2}$ remains fireable even after $k?\mathit{flow}.y$ has fired, and, if $[ID : 450, 500]^{H2}$ is fired, then both $k?\mathit{flow}.y$ *and* $[\mathit{AdjustValve} : 200, 300]^{H1}$ are disabled. Contrast this with the net for choice in Figure 4.13. □

Definition 4.23 Let $(W_i, \Theta_i, W_i^I) = \mathcal{N}[\![\widehat{P}_i]\!]$, for $i \in \{1, 2\}$, be disjoint nets. Then

$$\mathcal{N}[\![\widehat{P}_1 \,[> \widehat{P}_2]\!] \mathrel{\widehat{=}} (W_1 \cup W_2, \Theta, W_1^I \cup W_2^I)$$

where

$$\begin{array}{rcl} \Theta & \widehat{=} & \{\theta \mid \theta \in \Theta_1 \wedge \theta^\bullet \neq \{\mathsf{tick}\}\} \\ & \cup & \{({}^\bullet\theta, {}^\circ\theta \cup W_2^I, \alpha\theta, \theta^\bullet) \mid \theta \in \Theta_1 \wedge \theta^\bullet = \{\mathsf{tick}\}\} \\ & \cup & \{\theta \mid \theta \in \Theta_2 \wedge {}^\bullet\theta \notin W_2^I\} \\ & \cup & \{({}^\bullet\theta, {}^\circ\theta \cup W_1, \alpha\theta, \theta^\bullet) \mid \theta \in \Theta_2 \wedge {}^\bullet\theta \in W_2^I\} \end{array}$$

□

Parallel Composition

Control in a parallel composition, $\widehat{P}_1 \mid \widehat{P}_2$, is maintained independently in each process. Moreover, the parallel operator occurs only at the top-level, i.e. we never encounter terms such as $(\widehat{P}_1 \mid \widehat{P}_2)\ ;\ \widehat{P}_3$. In this case, the net for $\widehat{P}_1 \mid \widehat{P}_2$ can be constructed simply as the independent nets for $\widehat{P}_1$ and $\widehat{P}_2$.

Definition 4.24 Let $(W_i, \Theta_i, W_i^I) = \mathcal{N}[\![\widehat{P}_i]\!]$, for $i \in \{1, 2\}$, be disjoint nets. Then

$$\mathcal{N}[\![\widehat{P}_1 \mid \widehat{P}_2]\!] \mathrel{\widehat{=}} (W_1 \cup W_2, \Theta_1 \cup \Theta_2, W_1^I \cup W_2^I)$$

□

The benefits of the restricted use of parallelism can be seen here in the very simple net translation and in the fact that it is possible to support the translation of *bCANDLE* into nets in which every transition requires only a single trigger. The use of such simple nets has consequent benefits in the efficiency of the implementation of the TA construction which is based on them.

Process Variable

A process variable, X, represents a recursion point. As such, it has a net representation consisting of a place which is the trigger of a single transition whose target set is empty initially and is finalised later in the construction, on encountering the binding, $\mathrm{rec}\, X$. There is sure to be such a binding since we are dealing only with closed terms. Notice also that because of the restriction to systems with static control, a free process variable cannot be encountered on the left of a sequential composition and so the target set of the net for the process variable remains unchanged until the binding is encountered.

Definition 4.25 Let X be a process variable. The net of X is defined:

$$\mathcal{N}[\![X]\!] \mathrel{\widehat{=}} (\{w\}, \{(w, \{\}, X, \{\})\}, \{w\})$$

where $w \neq \mathsf{tick}$ is a place. □

Recursion Operator

The construction of the net for the recursion operator, $\mathrm{rec}\, X.\widehat{P}$, involves the resolution of the target sets of all those transitions in the net for $\widehat{P}$ whose attribute is the free process variable X. If (W, Θ, W^I) is the net constructed for $\widehat{P}$, then each such target set is made equal to the set, W^I, of initial places of $\widehat{P}$; i.e., the 'knot is tied'.

Example 4.9 Consider the term

$$\mathrm{rec}\, \mathit{Flow}.[\mathit{ReadSensor} : 85, 90]^{H1} \;;\; k!\mathit{flow}.x \;;\; \mathit{Flow}$$

which models control in a system which repeatedly reads a flow sensor, storing the reading in the variable x, and transmits its value on channel k. Its net is shown in Figure 4.15; the knot is tied simply in this case by returning control

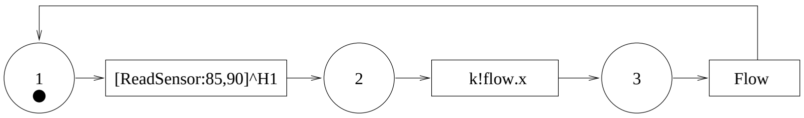


Fig. 4.15: Net for a recursion

to the beginning of the process. Notice that control is returned *indirectly* from $k!\mathit{flow}.x$ to the start of the process at place 1 via the transition *Flow* triggered by place 3. A more compact net can be used in which the redundant place and transition (place 3 and *Flow*) are omitted and in which control is returned directly from $k!\mathit{flow}.x$ to the beginning of the process (see Figure 4.16). It will

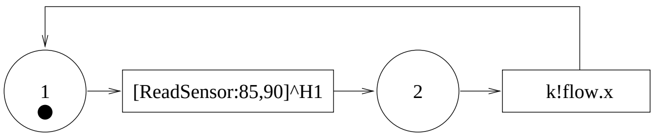


Fig. 4.16: Compact net for a recursion

be shown later how such indirections can be systematically removed and we will assume that this is always done in the nets which we construct. □

Definition 4.26 Let $\mathcal{N}[\![\widehat{P}]\!] = (W, \Theta, W^I)$, then the net of $\mathrm{rec}\, X.\widehat{P}$ is given by

$$\mathcal{N}[\![\mathrm{rec}\, X.\widehat{P}]\!] \mathrel{\widehat{=}} (W, \Theta', W^I)$$

where

$$\begin{aligned} \Theta' \;&\widehat{=}\; \{\theta \mid \theta \in \Theta \wedge \alpha\theta \neq X\} \\ \cup \;&\; \{({}^\bullet\theta, {}^\circ\theta, \alpha\theta, W^I) \mid \theta \in \Theta \wedge \alpha\theta = X\} \end{aligned}$$

□

Removing indirections

The net of a recursive process, constructed using the approach described above, contains places and transitions whose only purpose is to redirect the flow of control via a recursion point. Such transitions, which we have called *indirections*, have a process variable for their attribute. We can remove each of these transitions from the net, and also the places which trigger them, and transfer control directly to the start of the process. This avoids the generation of redundant locations and edges in the construction of the TA of the process. An algorithm is presented shortly which gives a method for the removal of indirections. First, we give an example which has been constructed to illustrate its most significant features.

Example 4.10 Consider the term

$$\widehat{P} \mathrel{\widehat{=}} \mathrm{rec}\, X.[a : 1]^{H1} \,;\, (X + \mathit{stop} \rightarrow [b : 0]^{H1} \,;\, \mathrm{idle})$$

which models a process which repeatedly executes an a action until the predicate *stop* becomes *true* when it executes a single b action and then idles forever. The net for this process, including indirections, is shown in Figure 4.17. For

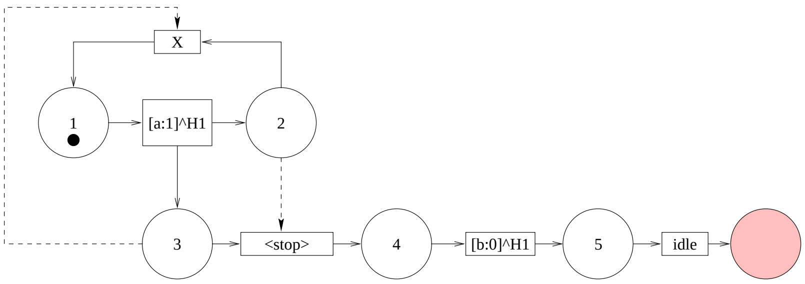


Fig. 4.17: A recursion with indirections

the most part, the net is unremarkable. But, notice that the unguarded process variable X, in the term $(X + \mathit{stop} \rightarrow [b : 0]^{H1} \,;\, \mathrm{idle})$, does not cause problems in the net representation: the term $\widehat{P}$ is represented by the net as shown, and the term $\widehat{P} + \mathit{stop} \rightarrow [b : 0]^{H1} \,;\, \mathrm{idle}$, which is reached after the recursion is unwound, is represented by the same net with the marking $\{1, 3\}$. The net contains a single indirection, namely the transition whose attribute is X and whose trigger is the place labelled 2. The result of removing this indirection is shown in Figure 4.18. In order to remove the indirection we need to perform the following steps:

1. Modify those transitions which are directed towards the indirection – in this case there is just one such transition, $[a : 1]^{H1}$ – so that they bypass the indirection and are directed to its target set instead – in this case, the place labelled 1.

2. Modify vulnerable sets to take account of the above change. There are two cases in which vulnerable sets need to be altered:

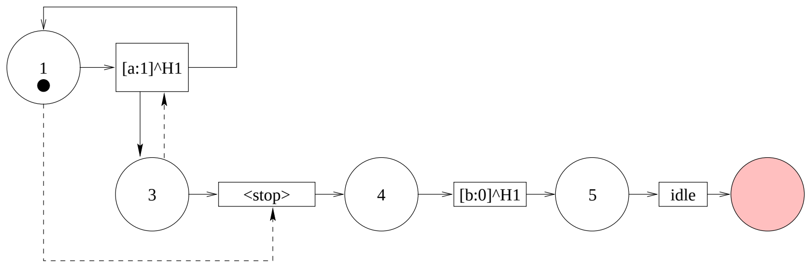


Fig. 4.18: A recursion with indirections removed

1 input
2   A net (W, Θ, W^I) constructed as described in §4.4.2
3 output
4   A new, equivalent net, (W', Θ', W^I), which does not contain indirections.
5 begin
6   $I := \{\, \theta \mid \alpha\theta = X, \text{for any process variable X} \,\}$
7   $W' := W \setminus \{\, {}^\bullet i \mid i \in I \,\}$
8   $\Theta' := \Theta \setminus I$
9   foreach $\theta \in \Theta'$ do
10   foreach $i \in I$ do
11   if ${}^\bullet i \in \theta^\bullet$ then $\theta^\bullet := \theta^\bullet \setminus \{{}^\bullet i\} \cup i^\bullet$ fi
12   if ${}^\bullet i \in {}^\circ\theta$ then ${}^\circ\theta := {}^\circ\theta \setminus \{{}^\bullet i\} \cup i^\bullet$ fi
13   if ${}^\bullet\theta \in i^\bullet$ then ${}^\circ\theta := {}^\circ\theta \cup {}^\circ i$ fi
14   od
15   od
16 end

Fig. 4.19: Algorithm to remove indirections

(a) If an indirection is vulnerable to some transition θ then all places to which it directs control should become vulnerable to θ. In Figure 4.17, 2 is vulnerable to $\langle stop \rangle$, so in the modified net (Figure 4.18) 1 has become vulnerable to this transition.

(b) Any places which are vulnerable to the indirection should instead be made vulnerable to those transitions to which control is directed by it. Notice in Figure 4.17 that 3 is vulnerable to X, whereas in the modified net of Figure 4.18, this place is vulnerable instead to $[a:1]^{H1}$. □

The algorithm in Figure 4.19 formalises a method for the removal of indirections. The following remarks are intended to explain this algorithm. I is the set of indirections, Θ' is the set of all transitions *except* indirections, W' is the set of all places *except* those which are the trigger of some indirection. For each transition $\theta \in \Theta'$ and for each indirection $i \in I$, the algorithm first causes the indirection to be bypassed (line 11); then all those places to which the indirection directs control are made vulnerable to θ, if i is vulnerable to θ (line 12); finally, if i

directs control to θ then all places vulnerable to i are made vulnerable to θ (line 13).

4.4.3 Final stage of timed automaton construction

The final stage of the construction of the TA for a system $(\widehat{P}, \widehat{N}, D)$ is to build the automaton itself, based on the net $\mathcal{N}[\![\widehat{P}]\!]$ constructed in the previous stage. If $\mathcal{R} = (W, \Theta, W^I)$ is the net for $\widehat{P}$, then a simple and efficient algorithm can be used to generate the TA for $(\widehat{P}, \widehat{N}, D)$ by starting from the initial location $(W^I, \widehat{N}, D)$ and visiting all reachable locations under the relation $\longrightarrow_{\mathcal{R}}$ as defined by rules **R.1** and **R.2** (Definition 4.18). A standard reachability algorithm is employed for this purpose (Figure 4.20). The following definition gives the details.

Definition 4.27 Let $(\widehat{P}, \widehat{N}, D) \in b\widehat{CAN}$ be a clocked *bCANDLE* system. Let $\mathcal{R} = (W, \Theta, W^I)$ be the net $\mathcal{N}[\![\widehat{P}]\!]$. Then, the TA $\mathcal{G}'(\widehat{P}, \widehat{N}, D) \mathrel{\widehat{=}} (Q, q^{\mathcal{I}}, A, \mathcal{H}, E, I)$ is built as follows:

- The set Q of locations is as given when the algorithm in Figure 4.20 terminates.
- The initial location $q^{\mathcal{I}}$ is $(W^I, \widehat{N}, D)$.
- The set A of action labels is $A_{\mathsf{p}} \cup A_{\mathsf{n}}$.
- The set $\mathcal{H}$ of clocks is the set of clocks associated with the attributes of the transitions in Θ, together with the network clocks and the urgent clock h_u, i.e., $\mathcal{H} \mathrel{\widehat{=}} \mathsf{clk}(W) \cup \mathsf{clk}(\widehat{N}) \cup \{h_u\}$, where $h_u \notin \mathsf{clk}(W) \cup \mathsf{clk}(\widehat{N})$.
- The set E of edges is as given when the algorithm in Figure 4.20 terminates.
- The invariant function $I : Q \to \Psi_{\mathcal{H}}$ is given by

$$\begin{aligned} I(W, \widehat{N}, D) &\mathrel{\widehat{=}} I(W, D) \wedge I(\widehat{N}) \\ I(W, D) &\mathrel{\widehat{=}} \bigwedge_{w \in W} I(\alpha\theta_w, D) \end{aligned}$$

  where $I(\widehat{\beta}, D)$ and $I(\widehat{N})$ are as in Definition 4.16 and

$$I(\langle\gamma\rangle, D) \mathrel{\widehat{=}} \mathsf{if}\ D \models \gamma\ \mathsf{then}\ h_u \leq 0\ \mathsf{else}\ \mathtt{tt}$$

□

For any clocked *bCANDLE* system $\widehat{B} \in b\widehat{CAN}$, we conjecture that $\mathcal{G}'(\widehat{B})$ is isomorphic to $\mathcal{G}(\widehat{B})$, and so, from Proposition 4.2, we conclude that its transition system is strongly equivalent to the corresponding *bCANDLE* system $\mathsf{unclk}(\widehat{B})$. The proof of the conjecture is left to future work.

input
  A *bCANDLE* system $(\widehat{P}, \widehat{N}, D)$
  A net $\mathcal{R} = (W, \Theta, W^I) = \mathcal{N}[\![\widehat{P}]\!]$
output
  The set of locations Q
  The set of edges E
begin
  $Q := \{(W^I, \widehat{N}, D)\}$
  $WAITING := \{(W^I, \widehat{N}, D)\}$
  $E := \emptyset$
  while $WAITING \neq \emptyset$ do
    remove some q from $WAITING$
    $E' := \{(q, \zeta, \lambda, \mathsf{H}, q') \mid q \xrightarrow{\zeta,\lambda,\mathsf{H}}_{\mathcal{R}} q'\}$
    $E := E \cup E'$
    foreach $(\_,\_,\_,\_, q') \in E'$ do
      if $q' \notin Q$
        add q' to Q
        add q' to $WAITING$
      fi
    od
  od
end

Fig. 4.20: Algorithm to construct a timed automaton

4.5 A simple example

In order to illustrate the automatic TA construction method, we return to the example of the simple flow regulator (§3.7). For ease of reference, the *bCANDLE* model is presented again in Figure 4.21. We briefly describe the various stages of the translation of the model to a TA.

Initially, the source file containing the model description is parsed, and equational definitions are rewritten as recursive process terms. Next, the clock variables are allocated. This gives the following clocked process term:

$$
\begin{array}{l}
((\text{rec } \mathit{Flow}.(([\mathit{ReadSensor} : 85, 90]^{H3};\ (k!\mathit{flow}.x;\ \text{idle})) \\
\quad [>([\mathit{PERIOD} : 10000, 10250]^{H5};\ \mathit{Flow}))) \\
| \\
(\text{rec } \mathit{Valve}.(k?\mathit{flow}.y;\ ([\mathit{AdjustValve} : 200, 300]^{H4};\ \mathit{Valve}))))
\end{array}
$$

The static components of each network channel are constructed from the details given in the network section of the model. A unique clock variable is allocated to each network channel. In this case, there is only one network channel k whose static components are:

- Clock $H2$,
- Message set $M = \{\mathit{flow}.\mathbf{1}\}$,
- Priority relation $\_ \prec \_ = \{\}$, and

```
Flow | Valve

where

Flow = [ReadSensor:85,90] ; k!flow.x ; idle
           [> [PERIOD:10000,10250] ; Flow

Valve = k?flow.y ; [AdjustValve:200,300]; Valve

network
/*          pri dlb dub dlB duB   */
  k = (flow : 1, 43, 53, 10, 12)

data x, y
```

Fig. 4.21: The flow regulator revisited

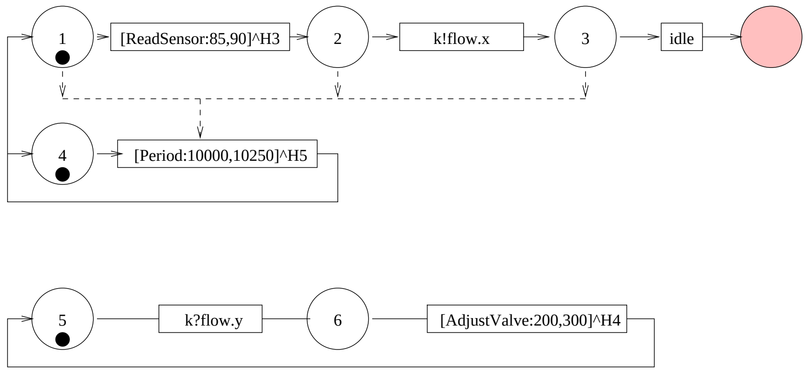


Fig. 4.22: Net for the flow regulator

- Transmission latency functions $\delta^{\mathsf{lb}} = \{\mathit{flow}.\mathbf{1} \mapsto 43\}$, $\delta^{\mathsf{ub}} = \{\mathit{flow}.\mathbf{1} \mapsto 53\}$, $\delta^{\mathsf{lB}} = \{\mathit{flow}.\mathbf{1} \mapsto 10\}$ and $\delta^{\mathsf{uB}} = \{\mathit{flow}.\mathbf{1} \mapsto 12\}$.

Clock $H1$ is used as the urgent clock.

The next stage of the translation is the construction of the net for the clocked process term. Figure 4.22 shows the net for our example.

Finally, the TA is constructed by applying the algorithm of Figure 4.20, starting from the initial location $(W, \widehat{N}, D)$, where the initial marking $W = \{1, 4, 5\}$; the initial state of the network $\widehat{N} = \{k \mapsto (\downarrow, \langle\rangle)\}$ i.e., the condition of channel k is *free* and its pending message queue is *empty*; and the initial data environment $D = \{x \mapsto \perp, y \mapsto \perp\}$.

The final TA has 48 locations, 146 edges, and uses 5 clock variables. It is shown in full in Appendix A. Many of the locations and edges in the generated TA are redundant, in the sense that some locations are unreachable and some edges are guarded by clock constraints which are unsatisfiable. This is typical of many automatic translators [Yov93, Bra95, Her98]. Since the clock constraints are not used to guide the construction of the TA, the worst-case complexity of the translation is comparable to that for an untimed language, i.e., exponential in the number of processes, channels and data variables. A more careful analysis of the clock constraints would allow many of the redundancies to be eliminated, although the fundamental complexity of the problem remains the same. This approach has not been implemented. Instead, it will be seen that an alternative approach presented in Chapter 5 addresses the problem in a way which appears to be effective in practice. Note also that it is possible to improve the quality of the generated TA by using a clock optimisation tool such as OptiKron [Daw98b], which produces an equivalent TA having a reduced number of clocks. For the example given here, OptiKron reduces the number of clocks required from 5 to 4.

Once the TA has been generated, a model-checking tool, such as KRONOS, can be used to ensure that the model exhibits desirable properties. For example, the simple bounded response property that the *AdjustValve* operation is always enabled within 300 time units of the enabling of the *ReadSensor* operation, can be expressed in TCTL as

$$init \Rightarrow \forall\Box(enable(ReadSensor) \Rightarrow \forall\Diamond_{\leq 300}\, enable(AdjustValve))$$

Let `flow.tctl` be a file containing a statement of this property in the syntax expected by KRONOS:

```
init IMPL AB (enable(OP_ReadSensor) IMPL
             (AD{<=300} enable(OP_AdjustValve)))
```

Let `flow.tg` be a file containing the TA generated from the *bCANDLE* model. The property can be checked in a forward reachability analysis using the command

```
kronos -forw flow.tg flow.tctl
```

giving the result

```
kronos: release 2.4.4 (i686) date Tue Aug 29 16:16:08 WET DST 2000
kronos: file flow.kro already exists
kronos: reading file flow.kro...
kronos: begin evaluation of flow.tctl
kronos: begin forward analysis
kronos: 14 simulation states generated
kronos: 14 simulation transitions generated
kronos: Invariance *** TRUE ***
kronos: end evaluation of flow.tctl
kronos: compacting

---------------------------------------------------------------------------
kronos: fixpoint          : system   0.010s * user   3.410s * #iterations 17
kronos: compact time      : system   0.000s * user   0.000s *
kronos: forward analysis  : system   0.000s * user   0.010s *
kronos: total time        : system   0.010s * user   3.460s *
---------------------------------------------------------------------------
```

Another property can be stated and checked, concerning the periodicity and jitter of the enabling of *AdjustValve*, for example,

$$\begin{aligned}&init \Rightarrow \forall\Diamond\, enable(AdjustValve) \wedge\\ &\quad \forall\Box(enable(AdjustValve) \Rightarrow\\ &\qquad \forall\Diamond_{\leq 100}((\forall\Box_{<9885}\neg\, enable(AdjustValve)) \wedge\\ &\qquad\qquad (\forall\Diamond_{<=10165}\, enable(AdjustValve))))\end{aligned}$$

which states that *AdjustValve* is eventually enabled and, whenever enabled, it fires within 100 time units, remaining disabled thereafter until it becomes enabled again after no less than 9885, and no more than 10165, time units. KRONOS verifies this property also.

The stated bounds (9885 and 10165) are as tight as possible for this example and, even for such a simple model, are not obvious by inspection. An analysis of this sort helps to build confidence in the quality of control which may be supplied by an implementation of the flow regulator.

In fact, there is a hidden assumption in the interpretation, given above, of the periodicity property: that whenever *AdjustValve* is enabled, it becomes disabled only by firing, and not as the result of an interrupt or timeout. In this case, the correctness of the assumption can be seen immediately by inspection of the model. However, in general, this may not be straightforward and one would like to be able to state the property in TCTL and check it using KRONOS. As Hernalsteen has observed [Her98], it is not so easy in TCTL to state properties concerning the firing of transitions as opposed to their enabling. This is because TCTL is a state-based, rather than an event-based, logic. The firing of a transition can be checked only by encoding this event somehow in the discrete state of the model. If the encoding is done by the modeller in an ad-hoc fashion, there is the possibility that errors will be introduced into the model; if it is done automatically for all events by the translator, the size of the state space will be increased, perhaps unnecessarily. One possible solution is to allow some events to be marked by the user for 'tracking' in the model, so that the translator can automatically add the encodings required only for those events of interest. Alternatively, one could consider the use of a logic in which both states and events can be referenced.

4.6 Conclusions

In this chapter, we have presented a translation to timed automata of the timed process language *bCANDLE*. The translation closely follows the semantic rules of the language and can be shown to be correct in a straightforward manner. We have also described an efficient method by which the translation can be implemented. The method adapts and extends techniques which have proved effective in similar settings [Gar92, Yov93]. A translator has been implemented and has been applied to a number of examples. As a result of this work, it is now possible, for the first time, to apply automatic analysis techniques, such as model checking, to system models which are described using a timed language which provides value-passing, prioritised, broadcast communication over latent channels as a primitive construct.

5. SPACE-EFFICIENT, ON-THE-FLY REACHABILITY ANALYSIS

5.1 Introduction

We have seen that reachability analysis and model checking of TA are well-established and successful techniques in the analysis of real-time systems. In Chapter 4, we have shown how *bCANDLE* models can be translated into TA, and thus have provided a way by which these verification methods can be applied to *bCANDLE* systems. As usual, the state space explosion problem is the major limiting factor in the use of such techniques, from a technical point of view. Much current research in TA verification is aimed at alleviating the worst effects of this problem: in particular, on-the-fly and symbolic approaches have proven effective in this respect. In this chapter, we consider how such methods can be adapted for use in the analysis of *bCANDLE* models.

Traditionally, a system model is presented as a network of small component TA, and on-the-fly methods, especially, derive their benefit from the fact that it may be unnecessary to construct their product automaton completely, before the verification question can be decided. Unfortunately, in the approach taken in Chapter 4, it is necessary to construct a monolithic TA for the system model as a whole, before verification begins. This is contrary to the spirit of on-the-fly verification, since, even though the verification problem may be decided during construction of the simulation graph, the monolithic TA itself may be very large. Recall that we need to build a monolithic TA if we wish to stay within the framework of the standard timed safety automata (TSA) of Henzinger et al. [HNSY94], since the product construction for TSA does not allow a satisfactory modelling of broadcast communication as we have defined it in Chapter 3. It may be possible to define a translation of *bCANDLE* models into networks of TA if we allow the use of TA extended with data variables and guards [ALST98, Boz98, LPY97]. This would allow us to take advantage of existing on-the-fly techniques. However, we do not pursue that line of enquiry in this work, but rather propose a novel solution to the problem: namely to generate the simulation graph of a system directly from the extended net created for the construction of its TA, but *without ever constructing the TA explicitly*. In this way, we obtain full on-the-fly verification for *bCANDLE*. Although several proposals have been published for the verification of real-time languages by means of translation to TA [DOY94, Her98, JM95, NSY91], we believe that this is the first time that the approach described here has appeared in the literature.

In addition, we combine on-the-fly verification with a *compact representation* of the state space. Binary decision diagrams (BDD's) have been used successfully, mainly in the analysis of hardware systems where the need for a compact representation of boolean functions is prevalent [Bry86]. However, the modelling of software systems commonly employs a richer set of data types and this fact motivates the investigation of different encodings of sets of states than by their characteristic functions. In this chapter, we consider the use of minimised deterministic finite state automata (MA's) [HP99] for the storage of the set of visited states in the reachability analysis of *bCANDLE* models. This state space representation promotes sharing of common parts of a set of state vectors, and seems to be particularly useful in mitigating the effects of state explosion caused by interleaving in asynchronous models. So far as we know, this is the first time that this state compression technique has been investigated in the analysis of timed systems.

The rest of this chapter is organised as follows: in §5.2 the algorithm for on-the fly reachability analysis is described; minimised automata are introduced in §5.3 and their use in the representation of the set of visited states is described in §5.4; salient features of an experimental platform are outlined in §5.5 and experimental results are discussed in §5.6; in §5.7, we consider related work; and finally, in §5.8, we present our conclusions and suggestions for further work.

5.2 On-the-fly reachability analysis

5.2.1 Basic algorithm

We consider the problem of determining whether or not it is possible for a given *bCANDLE* system (P, N, D) to reach a state which satisfies some state formula p. Recall that the validity of any state formula p can be determined locally for any state σ, and that $\sigma \Vdash p$ denotes the fact that σ satisfies p.

In fact, all the machinery needed to solve this problem is already in place. The algorithm for constructing a TA from a *bCANDLE* system is given in Figure 4.20 and the algorithm for reachability in the simulation graph of a TA is given in Figure 2.14. Clearly, we can solve our problem by executing these algorithms consecutively. However, it is straightforward to combine them into a single algorithm which solves the reachability problem without explicitly constructing the TA. Such an algorithm is shown in Figure 5.1, to which we refer in the following explanatory comments.

We assume that $(\widehat{P}, \widehat{N}, D)$ is a safely clocked *bCANDLE* system and that $c_{max}(\widehat{P}, \widehat{N}, D)$ gives the value of the largest constant appearing in a computation $[\omega : t_1, t_2]$ in $\widehat{P}$, or a message transmission latency bound $\delta^{\mathsf{ub}}(m)$, $\delta^{\mathsf{uB}}(m)$ in $\widehat{N}$. The net $\mathcal{R} = (W, \Theta, W^I)$, given by $\mathcal{N}[\![\widehat{P}]\!]$, is constructed as described in §4.4.2. We wish to check whether a state satisfying the state formula p is reachable from the initial state comprising the location $q^{\mathcal{I}} = (W^I, \widehat{N}, D)$ and the clock zone $\mathsf{suc}_{\mathcal{T}}^{q^{\mathcal{I}}}(\mathsf{zero})$. The rest of the algorithm shows a standard depth-first or breadth-first search of the reachable state space. The only section warranting further comment concerns the calculation of successor states at lines (14–15). Notice here that a location q is of the form $(W, \widehat{N}, D)$ and

input
  initial system $(\widehat{P}, \widehat{N}, D)$, $c = c_{max}(\widehat{P}, \widehat{N}, D)$
  net $\mathcal{R} = (W, \Theta, W^I) = \mathcal{N}[\![\widehat{P}]\!]$
  state formula p
begin
  $q^{\mathcal{I}} := (W^I, \widehat{N}, D)$
  $\mathit{VISITED} := \{(q^{\mathcal{I}}, \mathsf{suc}_{\tau}^{q^{\mathcal{I}}}(\mathsf{zero}))\}$
  $\mathit{WAITING} := \{(q^{\mathcal{I}}, \mathsf{suc}_{\tau}^{q^{\mathcal{I}}}(\mathsf{zero}))\}$
  while $\mathit{WAITING} \neq \emptyset$ do
    remove some (q, ζ) from $\mathit{WAITING}$
    if $(q, \zeta) \models p$
      then return 'yes'
      else
        $\mathit{succ} := \{(q', \zeta') \mid q \xrightarrow{\zeta'', \lambda, \mathsf{H}}_{\mathcal{R}} q' \wedge e = (q, \zeta'', \lambda, \mathsf{H}, q') \wedge$
            $\zeta' = \mathsf{close}_c(\mathsf{suc}_{\tau}^{q'}(\mathsf{suc}_e(\zeta))) \neq \emptyset\}$
        foreach $(q_s, \zeta_s) \in \mathit{succ}$ do
          if $(q_s, \zeta_s) \notin \mathit{VISITED}$
            add (q_s, ζ_s) to $\mathit{VISITED}$
            add (q_s, ζ_s) to $\mathit{WAITING}$
          fi
        od
      fi
    od
    return 'no'
  end

Fig. 5.1: Algorithm for on-the-fly reachability for *bCANDLE*

that the relation $q \xrightarrow{\zeta, \lambda, \mathsf{H}}_{\mathcal{R}} q'$ yields successor locations $q' = (W', \widehat{N}', D')$ according to the rules **R.1** and **R.2** (Definition 4.18). Each such successor determines an edge $e = (q, \zeta'', \lambda, \mathsf{H}, q')$ which can be used in the calculation of the clock zone successors of ζ in the usual way: $\zeta' = \mathsf{close}_c(\mathsf{suc}_{\tau}^{q'}(\mathsf{suc}_e(\zeta)))$. This allows the algorithm to follow the usual pattern for simulation graph reachability (Figure 2.14).

5.2.2 Clock activity reduction

The memory requirements of the basic on-the-fly reachability algorithm can be reduced considerably by reducing the number of clock variables which are used in the model to be analysed. Daws and Yovine [DY96] have proposed an important technique, known as *clock activity* reduction, which ensures that only the *active* clocks are recorded in each symbolic state in the simulation graph. A clock is considered to be active if its value will be *tested* before it is next *reset*. It is clear that there is no need to record the values of the other clocks, since they can have no effect upon the behaviour of the system until their current values have been destroyed by a clock reset. The remainder of this section shows how clock activity reduction can be employed in the on-the-fly reachability algorithm for *bCANDLE*.

Active clocks

Let $\mathcal{A}$ be a TA with set Q of locations, set $\mathcal{H}$ of clocks and set E of edges. Define an *edge path* of length n, over E, to be a sequence $\mathbf{e}$ of edges, $e_0, e_1, \ldots, e_{n-1}$, where $n \in \mathbb{N}$, $e_i \in E$, and, for $0 < i < n$, $\mathsf{src}(e_i) = \mathsf{tgt}(e_{i-1})$. Let E-path denote the set of edge paths over E, and $|\mathbf{e}|$ denote the length of edge path $\mathbf{e} \in E$-path.

We say that a clock $h \in \mathcal{H}$ is *tested* in location $q \in Q$, if h occurs in the invariant $I(q)$ or in the guard of some outgoing edge of q. We denote by $\mathsf{tclk}(q)$ the set of clocks tested in q. A clock h is said to be *active* in location q iff it is either tested in q or is tested in some location $q' \in Q$ which is connected to q by an edge path along which h is never reset.

Definition 5.1 (Active clocks) Let $\mathcal{A} = (Q, q^{\mathcal{I}}, A, \mathcal{H}, E, I)$ be a TA. Let $\mathsf{tclk} : Q \rightarrow 2^{\mathcal{H}}$ define, for each location $q \in Q$, the set of clocks occurring either in the invariant $I(q)$, or in the clock constraint of some outgoing edge of q. Then, for any location $q \in Q$, the set of *active clocks* of q is denoted $\mathsf{act}(q)$, and is defined by

$$\mathsf{act}(q) \mathrel{\widehat{=}} \mathsf{tclk}(q) \cup \overline{\mathsf{H}}(q)$$

where a clock h is in $\overline{\mathsf{H}}(q)$ iff h is not tested in q but is tested in some location connected to q by an edge path along which h is never reset, i.e.

$$\begin{aligned}\overline{\mathsf{H}}(q) \mathrel{\widehat{=}} \{h \in \mathcal{H} \mid h \notin \mathsf{tclk}(q) \wedge &\\ &(\exists\, \mathbf{e} \in E\text{-path}, q' \in Q \,.\, q = \mathsf{src}(e_0) \wedge q' = \mathsf{tgt}(e_{|\mathbf{e}|-1}) \wedge \\ &h \in \mathsf{tclk}(q') \wedge h \notin \textstyle\bigcup_{0 \le i < |\mathbf{e}|} \mathsf{reset}(e_i))\}\end{aligned}$$ □

Activity graph

An activity function $\mathsf{act} : Q \rightarrow 2^{\mathcal{H}}$ can be used in a *dimension-restricting* projection of the convex $\mathcal{H}$-polyhedron ζ, occurring in a symbolic state (q, ζ), in order to produce a new symbolic state (q, ζ'), where ζ' is a polyhedron on $\mathsf{act}(q) \subseteq \mathcal{H}$ instead of on $\mathcal{H}$. If $\mathsf{act}(q) \subset \mathcal{H}$, then the DBM representation of ζ' can be smaller than the representation of ζ. In practice, for a TA constructed from a *bCANDLE* description, as described earlier, the savings are usually very significant and allow the analysis of many models which would be intractable without this reduction.

Definition 5.2 (Dimension restricting projection [Tri98])
Given a $\mathcal{H}$-polyhedron ζ and a subset of clocks $\mathsf{H} \subseteq \mathcal{H}$, the *dimension-restricting* projection of ζ to H, denoted $\zeta\rfloor_{\mathsf{H}}$, is the H-polyhedron ζ' such that

$$\mathbf{v}' \in \zeta' \text{ iff } \exists\, \mathbf{v} \in \zeta \,.\, \forall\, h \in \mathsf{H} \,.\, \mathbf{v}(h) = \mathbf{v}'(h)$$ □

Definition 5.3 (Activity Graph [Tri98]) Let $\mathcal{A} = (Q, q^{\mathcal{I}}, A, \mathcal{H}, E, I)$ be a TA with $c \geq c_{max}(\mathcal{A})$. Let $\mathsf{act} : Q \rightarrow 2^{\mathcal{H}}$ be an activity function for $\mathcal{A}$. The *activity graph* of $\mathcal{A}$ with respect to c, starting at the symbolic state z_0, is denoted $\mathsf{AG}(\mathcal{A}, c, z_0)$, and is obtained from the simulation graph $\mathsf{SG}(\mathcal{A}, c, z_0)$ by the following modification:

- For each node (q, ζ) of $\mathsf{SG}(\mathcal{A}, c, z_0)$, the node $(q, \zeta\rfloor_{\mathsf{act}(q)})$ is a node of $\mathsf{AG}(\mathcal{A}, c, z_0)$
- For each edge $(q, \zeta)\xrightarrow{a}(q', \zeta')$ of $\mathsf{SG}(\mathcal{A}, c, z_0)$, $(q, \zeta\rfloor_{\mathsf{act}(q)})\xrightarrow{a}(q', \zeta'\rfloor_{\mathsf{act}(q')})$ is an edge of $\mathsf{AG}(\mathcal{A}, c, z_0)$. □

Notation. The activity graph of $\mathcal{A}$ with respect to c, starting at the initial state $(q^{\mathcal{I}}, \mathsf{zero})$, is denoted simply by $\mathsf{AG}(\mathcal{A}, c)$, and $\mathsf{AG}(\mathcal{A})$ denotes $\mathsf{AG}(\mathcal{A}, c_{max}(\mathcal{A}))$.

Tripakis [Tri98] shows that the activity graph preserves the same properties as the simulation graph. In particular, the correctness theorem (Proposition 2.6) is preserved, and so we can safely use the activity graph to decide reachability properties. In fact, it is trivial to modify the algorithm of Figure 5.1 to achieve this. We simply replace the calculation of successors (lines 14–15) so that each clock zone is restricted to the active clocks, as follows

$$\begin{aligned} succ := \{(q', \zeta'\rfloor_{\mathsf{act}(q')}) \mid\ & q \xrightarrow{\zeta'', \lambda, \mathsf{H}}_{\mathcal{R}} q' \wedge e = (q, \zeta'', \lambda, \mathsf{H}, q') \wedge \\ & \zeta' = \mathsf{close}_c(\mathsf{suc}_\tau^{q'}(\mathsf{suc}_e(\zeta))) \neq \emptyset\} \end{aligned}$$

Calculating active clocks in *bCANDLE*

In [DY96], an algorithm is given to compute the activity function act from the syntactic structure of a single TA modelling the entire system. It is shown in [DT98] how to compute and apply act on-the-fly, during construction of the simulation graph of the parallel composition of a set of TA. In order for activity reduction to be useful in the reachability analysis of *bCANDLE*, it is necessary to achieve a similar on-the-fly computation of act. We see from the modification above, to the on-the-fly algorithm for *bCANDLE*, that the only point at which act is required is during the calculation of successors, when, given an edge $e = (q, \_, \_, \_, q')$, we need to be able to compute $\mathsf{act}(q')$. Before considering the calculation of active clocks, we first identify the clocks which are tested in a given location.

A location q, in the TA of a *bCANDLE* system, is a tuple $(W, \widehat{N}, D)$. The set of *tested* clocks of such a location is defined below.

Definition 5.4 (Tested Clocks) Let $\widehat{B} \in b\widehat{CAN}$ be a clocked *bCANDLE* system and $\mathcal{G}'(\widehat{B}) = (Q, q^{\mathcal{I}}, A, \mathcal{H}, E, I)$ the TA constructed by Definition 4.27. Let $(W, \widehat{N}, D) \in Q$. The *tested* clocks of $(W, \widehat{N}, D)$ are denoted $\mathsf{tclk}(W, \widehat{N}, D)$, where

$$
\begin{array}{rcl}
\mathsf{tclk}(W, \widehat{N}, D) & \widehat{=} & \mathsf{tclk}(W, D) \cup \mathsf{tclk}(\widehat{N}) \\
\mathsf{tclk}(W, D) & \widehat{=} & \bigcup_{w \in W} \mathsf{tclk}(\alpha\theta_w, D) \\
\mathsf{tclk}(k!i.x, D) & \widehat{=} & \{h_u\} \\
\mathsf{tclk}(k?i.x, D) & \widehat{=} & \emptyset \\
\mathsf{tclk}([\omega : t_1, t_2]^h, D) & \widehat{=} & \mathsf{if}\ t_1 \in \mathbb{N} \vee t_2 \in \mathbb{N}\ \mathsf{then}\ \{h\}\ \mathsf{else}\ \emptyset \\
\mathsf{tclk}(\langle\gamma\rangle, D) & \widehat{=} & \mathsf{if}\ D \models \gamma\ \mathsf{then}\ \{h_u\}\ \mathsf{else}\ \emptyset \\
\mathsf{tclk}(\widehat{N}) & \widehat{=} & \bigcup_{k \in K} \mathsf{tclk}(\widehat{N}_k) \\
\mathsf{tclk}(\Downarrow, \langle\rangle)^h & \widehat{=} & \emptyset \\
\mathsf{tclk}(\Downarrow, m{:}u)^h & \widehat{=} & \{h_u\} \\
\mathsf{tclk}(\overset{t_1,t_2}{\rightsquigarrow} m, u)^h & \widehat{=} & \mathsf{if}\ t_1 \in \mathbb{N} \vee t_2 \in \mathbb{N}\ \mathsf{then}\ \{h\}\ \mathsf{else}\ \emptyset \\
\mathsf{tclk}(\uparrow m, u)^h & \widehat{=} & \{h_u\} \\
\mathsf{tclk}(m \overset{t_1,t_2}{\rightsquigarrow}, u)^h & \widehat{=} & \mathsf{if}\ t_1 \in \mathbb{N} \vee t_2 \in \mathbb{N}\ \mathsf{then}\ \{h\}\ \mathsf{else}\ \emptyset
\end{array}
$$

□

It is easy to see that a clock h appears in the invariant, or the guard of an outgoing edge, of a location $(W, \widehat{N}, D)$ iff $h \in \mathsf{tclk}(W, \widehat{N}, D)$. This follows directly from the definitions of tclk, rules **R.1** and **R.2** (Definition 4.18) and the invariant function (Definition 4.27).

Now we observe that, in fact, for any location $q = (W, \widehat{N}, D)$, the set of clocks *active* in q is identical to the set of clocks *tested* in q.

Proposition 5.1
Let $\widehat{B} \in b\widehat{CAN}$ be a clocked bCANDLE system and $\mathcal{G}'(\widehat{B}) = (Q, q^{\mathcal{I}}, A, \mathcal{H}, E, I)$ its TA. Then, for any $q \in Q$, it is the case that $\mathsf{act}(q) = \mathsf{tclk}(q)$.

Proof By definition, $\mathsf{act}(q) = \mathsf{tclk}(q) \cup \overline{\mathsf{H}}(q)$. We show that $\overline{\mathsf{H}}(q) = \emptyset$, for any $q \in Q$. The following lemma is required:

Lemma 5.1 *For any clock $h \in \mathcal{H}$ and any edge $e \in E$, if $h \notin \mathsf{tclk}(\mathsf{src}(e))$ and $h \in \mathsf{tclk}(\mathsf{tgt}(e))$, then $h \in \mathsf{reset}(e)$*

Proof Let $\mathsf{src}(e) = (W_1, \widehat{N}, D)$ and $\mathsf{tgt}(e) = (W_2, \widehat{N}', D')$. Observe that e must be derived using one of the rules **R.1** or **R.2**. We consider a clock $h \in \mathcal{H}$ such that $h \notin \mathsf{tclk}(W_1, \widehat{N}, D)$ and either $h \in \mathsf{tclk}(W_2, D')$ or $h \in \mathsf{tclk}(\widehat{N}')$. We show that $h \in \mathsf{reset}(e)$. There are two cases to consider.

(Case $h \in \mathsf{tclk}(W_2, D')$) By definition, $\mathsf{tclk}(W_2, D') = \bigcup_{w_2 \in W_2} \mathsf{tclk}(\alpha\theta_{w_2}, D')$. By **R.1**, we have that for all $w_2 \in W_2$, either there is some $w_1 \in W_1$ and transition $\theta_{w_1} = (w_1, W^V, \alpha, W^T)$ such that $w_2 \in W^T$, or $w_2 \in W_1 \setminus W^T$. If $w_2 \in W^T$ then, since $\mathsf{tclk}(W^T, D') \subseteq \mathsf{clk}(W^T) \subseteq \mathsf{reset}(e)$, we have $h \in \mathsf{reset}(e)$.

On the other hand, if $w_2 \in W_1 \setminus W^T$, then $h \equiv h_u$; this must be so since, by assumption, $h \notin \mathsf{tclk}(W_1, D)$, and, therefore, must be tested in D' by reason of the fact that $\alpha\theta_{w_2} = \langle\gamma\rangle$ for some data guard γ such that $D \not\models \gamma$ and $D' \models \gamma$. Since the urgent clock is reset on every edge, again, we have $h \in \mathsf{reset}(e)$.

(Case $h \in \mathsf{tclk}(\widehat{N}')$) If $h \in \mathsf{tclk}(\widehat{N}')$ then either $h \equiv h_k$, for some channel identifier k, such that $\widehat{N}_k = (\_, h_k)$, or $h \equiv h_u$. If $h_k \notin \mathsf{tclk}(\widehat{N})$ and $h_k \in \mathsf{tclk}(\widehat{N}')$ then e must be derived by **R.2** using either **E_N.1** or **E_N.3**. In both cases, $h \equiv h_k \in \mathsf{reset}(e)$. On the other hand, if $h \equiv h_u$, then $h \in \mathsf{reset}(e)$. In either case, the result follows.

□

Now, observe that $\overline{\mathsf{H}}(q) = \bigcup_{n \geq 0} \overline{\mathsf{H}}_n(q)$, where $\overline{\mathsf{H}}_0(q) \mathrel{\widehat{=}} \emptyset$, and, for $n > 0$,

$$\begin{aligned}\overline{\mathsf{H}}_n(q) \mathrel{\widehat{=}} \{h \in \mathcal{H} \mid h \notin \mathsf{tclk}(q) \wedge \\ &(\exists\, \mathbf{e} \in E\text{-path}, q' \in Q\ .\ |\mathbf{e}| = n \wedge q = \mathsf{src}(e_0) \wedge q' = \mathsf{tgt}(e_{n-1}) \wedge \\ &\quad h \in \mathsf{tclk}(q') \wedge h \notin \textstyle\bigcup_{0 \leq i < n} \mathsf{reset}(e_i))\}\end{aligned}$$

The proof of the proposition then follows by induction on the length of an E-path. □

This result has significant implications for the efficiency of the analyses which can be performed on *bCANDLE* systems, which surpasses that which can be achieved for general TA models where this property may not be exhibited. Most significantly, the result justifies the use of tclk as the activity function in the construction of the activity graph. Clearly, tclk can be calculated locally for any given location and, therefore, can be implemented efficiently and applied on-the-fly to achieve clock activity reduction during construction of the graph. The experimental data presented in §5.6 and §6.6.3 provides evidence for the utility of this technique in practice.

5.3 A Minimised Automaton Representation of Reachable States

Even with the clock activity reduction described in the previous section, the size of the state space, which arises in the analysis of a system model, can grow too big to be stored in computer memory. There are many proposals in the literature for reducing the memory required to store a set of states – see §5.7. In this section, we consider an approach in which a state vector is regarded as being encoded as a string over some alphabet, and the set of visited states is represented by a minimised deterministic finite automaton (MA) which recognises the language comprising the set of state vector strings. This technique has been implemented in the model checker SPIN [HP99], where it has been shown to achieve even better compression for many systems than that obtained by the use of BDD's [Vis96]. Similar results have been reported in experiments using *sharing trees* [GGZ95, Zam97] (also known as *GE-SETS* [Gré96]); the implementation of the sharing tree data structure is very similar to the MA implementation described here. By comparison with other techniques which also ensure complete state space coverage, the space reductions achieved by the use of MA's are among the best reported in the literature. It is of considerable interest to see if this performance is observed also in the storage of the state spaces which arise in the analysis of timed systems. Our work is the first to report such experiments.

In the remainder of this section, we introduce the basic ideas and definitions for the use of MA's in state space storage. Later, we discuss their application in the implementation of a state store for *bCANDLE*.

5.3.1 Minimised Deterministic Finite State Automata

Definition 5.5 A k-layer *deterministic finite state automaton* (DFA) is a tuple $\mathcal{A} = (Q, A, E)$ where

- $Q = \bigcup\{Q_i \mid 0 \leq i \leq k\}$ is the set of states. Q_i, $\emptyset \subset Q_i \subset Q$, is the set of states at the ith layer and $Q_i \cap Q_j = \emptyset$ for $i \neq j$. Q_0 is a singleton containing the initial state and $Q_k = \{\mathrm{T}, \mathrm{F}\}$, where T is the accepting final state and F is the rejecting final state. The set $Q \setminus Q_k$ is denoted Q^-.
- A is the alphabet.
- $E : Q^- \times A \rightarrow Q$ is a total function such that for all states $q \in Q^-$ and symbols $a \in A$, if $q \in Q_i$ then $E(q, a) \in Q_{i+1}$. □

A *string* $\mathbf{a}$ of *length* n is a sequence of symbols $\mathbf{a} = a_0, a_1, \ldots, a_{n-1}$, where $a_i \in A$ for $0 \leq i < n$. A^n denotes the set of strings of length n over the alphabet A. A string $\mathbf{a} = a_0, a_1, \ldots, a_{n-1}$ *generates a state sequence* $q_0, q_1, \ldots, q_n$ from state $q = q_0$, where $q_{j+1} = E(q_j, a_j)$ for $0 \leq j < n$. For a state $q \in Q_i$, we denote the *language of* q by $\mathcal{L}_{\mathcal{A}}(q)$. $\mathcal{L}_{\mathcal{A}}(q)$ is the set of strings which generate a state sequence from q ending with the terminal state T. Formally, for $q \in Q_i$,

$$\begin{aligned}\mathcal{L}_{\mathcal{A}}(q) \mathrel{\widehat{=}} \{&\mathbf{a} \in A^{k-i} \mid \mathbf{a} \text{ generates the state sequence } q_0, q_1, \ldots, q_{k-i} \\ &\text{from } q, \text{ and } q_{k-i} = \mathrm{T}\}\end{aligned}$$

We define $\mathcal{L}(\mathcal{A}) = \mathcal{L}_{\mathcal{A}}(q_0)$ where $q_0 \in Q_0$. A DFA is *minimised* provided $\mathcal{L}(q_i) = \mathcal{L}(q_j)$ iff $q_i = q_j$.

Example 5.1 The MA of Figure 5.2 is $\mathcal{A} = (\{Q_i\}_{i=0}^{4}, A, E)$, where $Q_0 = \{0\}, Q_1 = \{1, 2, 3\}, Q_2 = \{4, 5\}$, $Q_3 = \{6, 7\}$ and $Q_4 = \{\mathrm{T}, \mathrm{F}\}$ is the set of states; $A = \{a, b, c\}$ is the alphabet; and E is the set of edges as shown in the figure. $\mathcal{A}$ represents the set $S \subseteq A^4$ of strings where

$$\begin{aligned} S \;=\; \{&aaaa, aaba, aaca, abaa, abba, abca, acaa, acba, acca, \\ &baab, baba, baca, bbab, bbba, bbca, bcaa, bcba, bcca, \\ &caab, caba, caca, cbaa, cbba, cbca, ccab, ccba, ccca\} \end{aligned}$$

□

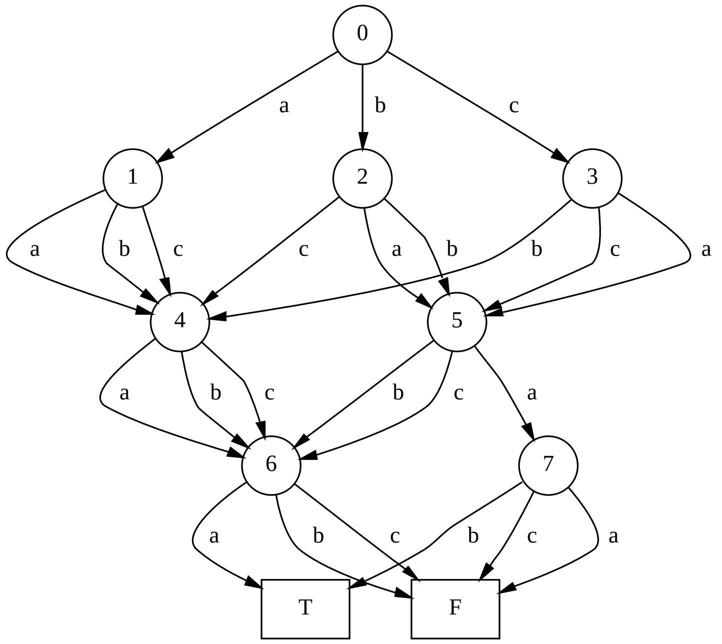


Fig. 5.2: A minimised automaton

It can be seen that a MA achieves a compact representation of a set of strings by a combination of *prefix merging* and *suffix merging*. The requirement that a MA is deterministic ensures that all shared prefixes are recorded only once. Similarly, the requirement of minimisation ensures that many shared suffixes are also recorded only once. Furthermore, for any MA, the amount of sharing in prefixes and suffixes is optimal, in the sense that any other MA recording the same information is guaranteed to be isomorphic. In fact, a MA gives a canonical representation of a language – there is only one MA (up to isomorphism) representing a given language, and different languages are represented by different MA's [HU79]. An effective use of MA's for state space representation will require that state vectors are organised so as to promote as much prefix and suffix merging as possible. It is worth noting that the compactness of sharing trees (GE-SETS) relies on the same idea. The relationship between MA's and sharing trees is discussed in [Zam97].

MA Operations

There are three basic operations on MA's which are required to implement a state store for reachability analysis:

initialise – create a MA $\mathcal{A}$ having an empty language, i.e., ensure $\mathcal{A}$ satisfies $\mathcal{L}(\mathcal{A}) = \emptyset$;

insert – given a MA $\mathcal{A}$ and a string $\mathbf{a}$, create a MA $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A}) \cup \{\mathbf{a}\}$;

LOCATION (q)	Marking (W)	$\{w_1, w_2, \ldots, w_m\}$
	Network ($\widehat{N}$)	$\{(s_1, u_1), (s_2, u_2), \ldots, (s_n, u_n)\}$
	Data (D)	$\{v_1, v_2, \ldots, v_d\}$
ZONE (ζ)		$\{b_1, b_2, \ldots, b_z\}$

Fig. 5.3: Structure of a *bCANDLE* state vector

member – given a MA $\mathcal{A}$ and a string $\mathbf{a}$, return *true* if $\mathbf{a} \in \mathcal{L}(\mathcal{A})$, otherwise return *false*.

Holzmann and Puri [HP99] give efficient algorithms for each of these operations. To be precise, for a k-layer MA $\mathcal{A}$ over an alphabet A, their **insert** algorithm is $O(k|A|)$, **member** is $O(k)$ and **initialise** is $O(1)$. We refer the reader to the cited work for a detailed description of the algorithms.

5.4 Implementing a MA state store for *bCANDLE*

In order to use a MA for the state store in a reachability analysis of a *bCANDLE* system, it is necessary to partition the state vector and allocate partitions to layers in the MA. How this is done can have a significant effect upon the efficiency of the state store. In this section, we discuss the structure of a *bCANDLE* state vector and consider some principles which may be applied in determining an effective partitioning.

5.4.1 The state vector

A *bCANDLE* state vector has the general form shown in Figure 5.3, where it can be seen that a state vector represents a location $q = (W, \widehat{N}, D)$ and a clock zone ζ. The representation is discussed in more detail below.

Marking The marking $W = \{w_1, w_2, \ldots, w_m\}$ is the set of marked places of the system net (§4.4.1) which represents the state of the system processes.

Network For each channel in the system network, its dynamically changing components are recorded in the state vector, i.e., the status s and the queue u of messages pending transmission. The status consists of one of the four values $\mathsf{FREE}, \mathsf{PRE}, \mathsf{ACCEPT}$, or POST, and, optionally, an associated message, comprising a message identifier and a data value. The message queue can be modelled in a variety of ways. In the following, we will assume a fixed length sequence of messages.

Data Let $Var = \{x_1, x_2, \ldots, x_d\}$ be the set of data variables, where each variable x_i ranges over a domain of values, V_{x_i}. The data environment D is represented by recording its valuation function $\mathsf{val} = \{x_1 \mapsto v_1, x_2 \mapsto v_2, \ldots, x_d \mapsto v_d\}$. This is done simply by fixing the order of the variables and storing the corresponding values $\{v_1, v_2, \ldots, v_d\}$.

M	0	1	2	3
0	4	$(-3, \leq)$	$(-5, \leq)$	$(-2, \leq)$
1	$(7, \leq)$	h_2	$(2, \leq)$	$(5, \leq)$
2	$(6, \leq)$	$(3, \leq)$	h_5	$(4, \leq)$
3	$(8, \leq)$	$(5, \leq)$	$(3, \leq)$	h_7

(a)

M'	0	1	2	3
0	3	$(-3, \leq)$	$(-5, \leq)$	$\perp$
1	$(7, \leq)$	h_2	$(2, \leq)$	$\perp$
2	$(6, \leq)$	$(3, \leq)$	h_5	$\perp$
3	$\perp$	$\perp$	$\perp$	$\perp$

(b)

M''	0	1	2
0	3	$(-3, \leq)$	$(-5, \leq)$
1	$(7, \leq)$	h_2	$(2, \leq)$
2	$(6, \leq)$	$(3, \leq)$	h_5

(c)

Fig. 5.4: Simple DBMs

Zone The clock zone ζ is represented as a DBM (§2.7.5), i.e. a set of bounds $\{b_1, \ldots, b_z\}$. Here, the main issue is how to take advantage of clock activity reduction in order to reduce the storage requirements. For example, consider a TA $\mathcal{A}$ with clock set $\mathcal{H} = \{h_1, h_2, \ldots, h_7\}$, whose set of reachable states contains no state in which more than 3 clocks are active simultaneously, and many states which have fewer than 3 active clocks. It is sensible to take advantage of this observation in the state vector representation of the DBM's. We illustrate this with an example. Figure 5.4(a) shows a DBM in which only the clocks h_2, h_5 and h_7 are active. First, notice that the diagonal of any DBM contains redundant information: for any $\mathcal{H}$-polyhedron ζ, $h - h = 0$, for all $h \in \mathcal{H}$. So this information need not be stored explicitly in a DBM representing ζ. Instead, we can use the diagonal to store the size of the (active part) of the DBM and the names of the active clocks, whose differences are represented in the DBM. In Figure 5.4(a), the value of $M_{0,0}$ indicates that M is a DBM of size 4; the value of $M_{1,1}$ shows that row (1) and column (1) represent clock h_2; the value of $M_{2,2}$ shows that row (2) and column (2) represent clock h_5; and so the value of $M_{1,2}$ shows that $h_2 - h_5 \leq 2$.

In representing a zone containing fewer than 3 active clocks, we can use a DBM of the same size as that for a 3-clock zone, marking as unused those cells which are not required. Figure 5.4(b) shows a DBM M' which represents a zone having 2 active clocks, h_2 and h_5. $M'_{0,0}$ shows that the active part of the DBM has size 3. We use $\perp$ to show that the entries in row (3) and column (3) are unused. In this way we can use DBMs of constant size in all states. This is useful in conjunction with a MA state store, where all state vectors are required to be the same length. Naturally, we choose the smallest size which is large enough to represent the maximum number of active clocks occurring in any state.

Cell #	0	1	2	3	4	5	6	7
Data	4	$(-3, \leq)$	$(-5, \leq)$	$(-2, \leq)$	$(7, \leq)$	h_2	$(2, \leq)$	$(5, \leq)$
Cell #	8	9	10	11	12	13	14	15
Data	$(6, \leq)$	$(3, \leq)$	h_5	$(4, \leq)$	$(8, \leq)$	$(5, \leq)$	$(3, \leq)$	h_7

Fig. 5.5: State vector representation of a 3-clock zone

Alternatively, we can choose to use DBMs of *variable dimension*, in which, for each state, there are only as many entries as are required to store the values of the active clocks for that state [Tri98]. Figure 5.4(c) shows the DBM M'', which represents the same zone as M' but has fewer entries. This representation can reduce memory requirements when clock zones are stored in an auxiliary hash table, and only a pointer to a clock zone is stored in each state vector entry in the MA.

In the following, we will use DBMs of both constant and variable dimension. Figure 5.5 shows a typical state vector representation of the DBM M.

5.4.2 Mapping the state vector to MA layers

It is clear that the way in which a state vector is partitioned, and the partitions allocated to layers, will have a major impact on the memory reduction achieved when storing a set of state vectors in a MA. Consider an extreme case in which the whole state vector is allocated to a single layer. All possibility for sharing is lost and there is no compensation for the overheads of implementing the MA. At the other extreme, one can consider a bit-level allocation, in which each bit of the state vector is allocated to a layer in the MA, giving, for a state vector of n bits, a MA of $n + 1$ layers over the alphabet $\{0, 1\}$. This scheme allows the possibility of maximal sharing, but increases the overheads incurred in implementing the layers: each bit of the state vector needs 2 pointers to encode it. It is easy to envision similar schemes with a different unit of allocation: byte or word, for example. The *height* of a MA is given by the number of layers, the *width* is the largest number of nodes on a layer, and is proportional to the size of the alphabet, $|A|$. A small unit of allocation leads to a 'tall, thin' MA, a large unit of allocation to a 'short, fat' MA. It is not clear, analytically, which scheme will lead to the most compact encoding, in general. Holzmann and Puri [HP99] show experimental data suggesting that a byte-level partitioning is a reasonable choice for the state spaces which they consider.

Another approach to partitioning the state vector seeks to maintain the integrity of state variables within a single layer of the MA. A simple application of this idea is a partitioning in which each variable is allocated to its own MA layer. A modification of this approach allows variables over small domains to be clustered together in a single layer. Only when the domain of a variable is considered to be too large, is it split over two or more layers. This approach is adopted effectively in the GE-SET implementation of [Gré96]. However, it is not so easy to implement this partitioning automatically, and the byte-level partitioning of [HP99] appears to be just as effective.

5.4.3 Variable Ordering

In common with other compact encodings, such as BDD's [Bry86] and GE-sets [Gré96], MA's are sensitive to variable ordering, i.e., the size of a MA representing a set of state vectors can be affected by the ordering of variables within the state vector: for some variable orderings, growth in memory usage may be linear in the number of states; for others, growth may be exponential. In the construction of MA's, the following guidelines have proved useful in achieving an acceptable growth.

- Ensure that the least frequently changing components of the state vector occur as prefixes and suffixes, in order to promote as much sharing as possible.
- Group together variables which are strongly related, i.e., which show clearly identifiable patterns of recurrence in the set of reachable state vectors.

These ideas have been confirmed frequently in applications of sharing trees [GGZ95, Gré96, Zam97], and the latter idea is familiar also to users of BDD's, where it arises in the well-known recommendation to interleave the variables of the pre- and post-states in representing a transition relation [Bry92].

In applying these principles to the construction of a MA state store for *bCANDLE*, we have considered the following possibilities:

- permutations of the major state vector components: marking W, context C (network and data environment) and zone Z.
- placement of cells within the encoding of the DBM representing the clock zone (see Figure 5.6):
  - **O0** is the standard row-major matrix encoding;
  - **O1** removes clock names from the diagonal, stores them before all other cells and then follows row-major ordering of the remaining cells;
  - **O2** removes clock names from the diagonal, as for **O1**, and additionally, stores contiguously both the lower and upper bounds for each clock difference.

Of course, there is no guarantee that this framework leads to the discovery of an optimal variable ordering. However, the experimental results indicate that, in many practical applications, it does lead to the discovery of an ordering whereby substantial reductions in memory requirements can be achieved.

Cell #	0	1	2	3	4	5	6	7	8
Data	3	$(-3, \leq)$	$(-5, \leq)$	$(7, \leq)$	h_2	$(2, \leq)$	$(6, \leq)$	$(3, \leq)$	h_5

(**O0**)

Cell #	0	1	2	3	4	5	6	7	8
Data	3	h_2	h_5	$(-3, \leq)$	$(-5, \leq)$	$(7, \leq)$	$(2, \leq)$	$(6, \leq)$	$(3, \leq)$

(**O1**)

Cell #	0	1	2	3	4	5	6	7	8
Data	3	h_2	h_5	$(-3, \leq)$	$(7, \leq)$	$(-5, \leq)$	$(6, \leq)$	$(2, \leq)$	$(3, \leq)$

(**O2**)

Fig. 5.6: Orderings of the cells of DBM M'' (see Figure 5.4)

5.5 An experimental platform

No state storage technique can escape the known worst-case complexity of reachability analysis. The best that can be hoped is that heuristics are identified which improve performance on a range of examples which arise in practice. This can only be confirmed empirically. In this section, we introduce the salient features of an experimental platform which has been developed in order to allow us to explore a variety of approaches to the analysis of *bCANDLE* systems.

5.5.1 The *bCANDLE* Compiler

We have implemented a prototype 'compiler' for *bCANDLE*. The compiler is written in ML and generates C code to perform a given task on a system model. The user can choose to

1. generate the timed automaton for the model (in KRONOS `.tg` format),
2. perform reachability analysis of the simulation graph on-the-fly, without first generating the timed automaton,
3. explore the simulation graph interactively.

It is the second facility which is of interest here. An important feature of the reachability analyser is that it provides the user with a wide choice of techniques for the storage of the state space.

5.5.2 State Space Storage Modes

The *bCANDLE* compiler is able to generate C code for a variety of state space storage modes. Each mode is based upon essentially the same state vector encoding, as introduced below.

State vector encoding

A state vector comprises a marking, network, data environment and clock zone.

Marking The marking is represented by a bitmap in which, for each place in the control system net, there is a corresponding bit, whose value is 1 if the place is marked and 0 if it is not.

Network The network is represented by a pair of arrays: a channel status array and a message array. The channel status array records the status of each channel, where a channel status is encoded in two 16 bit integers. Two bits of the first integer are allocated for the representation of the condition of the channel ($\mathsf{FREE}, \mathsf{PRE}, \mathsf{ACCEPT}$ or POST), 11 bits store the message identifier, and the remaining bits are unused. The second integer records the message value. If the channel condition is FREE, the message identifier and value are unused. The message array is a fixed length array of messages where each message is encoded in two 16 bit integers. Five bits of the first integer are used for the channel identifier, and the remaining 11 bits for the message identifier. The second integer records the message value. The length of the message array is user-definable, the optimal length being the maximum number of messages which are pending transmission simultaneously in the network of some system state.

Data environment The data environment $\{x_1 \mapsto v_1, x_2 \mapsto v_2, \ldots, x_d \mapsto v_d\}$ is encoded by fixing an order for the data variables and storing the values only. A data value $v_i \in V_{x_i}$ is encoded using $n_i = \lceil \log(|V_{x_i}|) \rceil$ bits, and the set of values is represented by $\lceil (\Sigma_{x_i \in Var} n_i)/8 \rceil$ bytes.

Zone The zone is represented by an array of bounds. Each bound $(c, \prec) \in \mathbb{Z}_\infty \times \{<, \leq\}$ is encoded using a 16 bit integer in which 15 bits are used for the constant value c and 1 bit to distinguish between the comparison operators, $<$ and $\leq$.

Storage modes

The following storage modes are defined:

H The state vector encoding is as described above. In each state, storage is allocated for the maximum number of clocks required system-wide, even though there may be many states in which fewer clocks are active (§5.2.2). The set of visited states is stored in a single hash table. This mode corresponds to the ‘naive’ approach, as adopted in early versions of UPPAAL and KRONOS.

M As for **H**, except that the set of visited states is stored as a MA. In constructing the MA, the state vector is treated as a string of bytes. A state vector of n bytes gives rise to a MA of $n + 1$ layers. In this mode, the user has further options to control the ordering, within the state vector, of the location components: marking W, context C (network and data environment) and zone Z. Any permutation of the location components

System	Procs	Vars	Chans	Mtypes	Clocks	Zones	States
Boiler1	2	2	1	1	5	22824	115660
Boiler2	2	3	1	1	3	588	1110198
Disbmut	4	10	1	6	6	46561	223604

Tab. 5.1: Test systems

is permissible. In addition, it is possible to choose one of the orderings **O0**, **O1** and **O2**, which modify the placement of cells within the encoding of the DBM representing the clock zone, as discussed in §5.4.3.

HV Each state vector is encoded as for mode **H**, except that the clock zone is represented by a pointer to a variable dimension matrix. The set of state vectors is stored in one hash table and the associated variable dimension matrices in another. This means that for each state, only sufficient storage is allocated for the number of clocks active in it, and that only one copy of each distinct clock zone is stored for the whole system. This mode corresponds closely to the storage method adopted in the most recent implementations of KRONOS [Tri98].

MV As for **HV**, except that the hash table storing the state vectors is replaced by a MA, constructed as for mode **M**.

For any of these storage modes, the user can choose whether or not to apply the clock activity reduction of §5.2.2.

5.6 Experiments

5.6.1 System models

We have tested our implementation on some example system models: *Boiler1* and *Boiler2* model part of a boiler control system; *Disbmut* models a CAN implementation of a standard algorithm for distributed mutual exclusion [Tan92]. The model included here is for a single coordinator and three competing processes. Table 5.1 gives information regarding the scale of the examples: number of processes, variables, CAN channels, message types and clocks required by each system, and the number of distinct zones and symbolic states identified in generating the whole of the reachable state space in the simulation graph.

5.6.2 Experimental results

Performance measurements for each system and each state space storage mode are given in Table 5.2. We show the time taken and the total memory used in generating the reachable state space. We take mode **H** as the basis of comparison and show memory compression and time overheads as percentages of the requirements of mode **H**. It should be noted that clock activity reduction was applied in all cases. The measurements were taken on a 233MHz Pentium II

System	Mode	Mem (Mb)	Comp %	Time (s)	Over %
Boiler1	H	11.90	100	15	100
	M	7.89	66	73	486
	HV	5.48	46	15	100
	MV	2.99	25	24	160
Boiler2	H	56.45	100	89	100
	M	3.79	7	205	230
	HV	25.44	45	81	92
	MV	3.30	6	140	157
Disbmut	H	34.53	100	66	100
	M	15.10	44	266	403
	HV	16.02	46	65	98
	MV	9.55	28	89	135

Tab. 5.2: Comparison of storage modes

System	Mode	Order	Nodes	Edges	Mem (Mb)
Boiler1	M	ZWC, O1	140757	429127	7.89
	M	WZC, O2	310224	879807	15.57
	MV	WZC	4659	57194	2.99
	MV	CZW	4420	73846	3.61
Boiler2	M	CWZ, O0	14002	49372	3.79
	M	ZCW, O0	54041	139680	5.33
	MV	CWZ	4988	28657	3.30
	MV	ZCW	45674	120875	5.04
Disbmut	M	CWZ, O1	258518	718351	15.10
	M	WZC, O2	–	–	>63.43
	MV	CWZ	74085	376663	9.55
	MV	WCZ	209094	651873	15.59

Tab. 5.3: Impact of variable ordering on minimised automaton modes

having 64Mb RAM (58Mb available) and 128Mb swap, running RedHat Linux 5.0.

Table 5.3 shows the state space usage of the variable orderings which show the best, and the worst, performance for each system and each MA mode. The nodes (resp. edges) column shows the total number of nodes (resp. edges) used in the final MA.

5.6.3 Discussion of experimental results

Reference to Table 5.2 shows that the most economical use of space, for all examples, involves the use of a MA. The memory reductions achieved range approximately from a factor of 4 to a factor of 17, with reductions at the lower end of this range for realistic systems. When considering the effect of the inclusion of timing information, we observe that the inclusion of the zone encoding in the MA (mode **M**) gives a worse use of space than that given by the use of variable dimension matrices (mode **MV**). This suggests that a MA representation does not enable sufficient sharing to compensate for the inclusion of redundant bounds within clock zones, nor does it allow for significant sharing of bounds, either in a union of zones associated with a single discrete state, or between zones associated with different discrete states. However, it is clear that the MA representation *is* effective in encoding the discrete state variables. Moreover, the inclusion of timing information, in the form of variable dimension matrices, although having a somewhat adverse effect, allows for significant memory reductions – compare the average reduction factor of 6.5, achieved for the timed systems analysed here, with that of 7.1 for systems without timing information as reported in [HP99].

As expected, we pay a time performance penalty for the use of MA, the average increase being by a factor of about 1.5. Notice, however, that in all cases, it is possible to find a MA mode in which the memory reduction significantly outweighs the time overhead.

Achieving good compression requires the use of a good variable ordering. From Table. 5.3, we observe that in 4 of 6 cases the best ordering for the state vector components is *context, marking, zone* (**CWZ**), and in 2 of 3 cases the best ordering for cell elements is **O1**. Although not shown here, we note that in the exceptional cases, the performance of **CWZ** and **O1** is only slightly worse than the best. Notice, however, that use of a bad variable ordering can be disastrous, as witnessed by the worst case[1] for *Disbmut*, mode **M**, which *increases* memory requirements by a factor of more than 2.

5.7 Related work

Courtiat and de Oliveira have proposed a similar approach to on-the-fly reachability analysis of a timed process algebra [CdO95]. Our approach has been developed independently and differs in several important respects. Firstly, it keeps within the framework of timed safety automata, which have been studied extensively [HNSY94, NSY91, NSY92, Sok96, Yov93, Yov97] and are well-understood; in their paper, Courtiat and de Oliveira propose a different model, called Dynamic Timed Automata, which appears not to be used elsewhere. Secondly, it is able to take advantage of a standard clock reduction technique [Daw98a, DT98, DY96, Tri98]. Finally, it is based upon a compact net representation of control states, which allows each state vector to be encoded very efficiently. By contrast, the method of Courtiat and de Oliveira uses structural configurations which are closely related to the abstract syntax of process terms, and consequently appears to suffer from a bloated state vector representation [ACdP97]. However, an interesting feature of their work is its use of the algorithm of Yannakakis and Lee [YL93] to minimise the reachability graph as it is constructed. We intend to investigate whether our approach can

[1] In fact, the worst case for *Disbmut* failed to terminate with the available resources; hence the approximation shown in the table.

benefit from this idea.

The use of binary decision diagrams (BDDs) for compact state space representation is well-known [BRB90, Bry86, Bry92]. However, there is a growing body of evidence which supports the view that, in the analysis of asynchronous systems, explicit state enumeration, combined with other compaction methods, often provides better performance – see for example the paper of Visser [Vis96] in which BDDs are compared unfavourably with sharing trees for representing the state space in the SPIN model checker.

If one is prepared to allow a very small probability that not all reachable states are considered in an analysis, then the bitstate technique of Holzmann [Hol95] and the probabilistic hash compaction of Stern [Ste97] can achieve memory reductions of one or two orders of magnitude.

The stack storage method of [Hol90] allows reachability analysis without the need to store the set of visited states at all, but at the cost of a potentially exponential increase in run time. Run time performance can be improved by adapting this technique with the maintenance of a state space cache [Hol85, JJ91]. This method is particularly effective in combination with partial order reduction [GHP95].

Representation of timing constraints by DBMs was proposed by Dill [Dil89] and has been preferred in the most efficient verification tools for timed systems, such as KRONOS [HNSY94] and UPPAAL [LPY97].

Wong-Toi and Dill [WTD94] and Balarin [Bal96] have each shown techniques for encoding the transition relation of timed systems using BDD's, approximating unions of zones using convex hulls. Bozga et. al. [BMPY97] offer a canonical representation of discretised sets of clock configurations using NDDs[2] [ABK+97], which are a BDD-based encoding amenable to combination with a symbolic representation of the discrete part of the system. The difficulty with these techniques is that they are very sensitive to the size of the constants in the timing constraints of the system model. If the constants are large then state space explosion is not controlled effectively.

Larsen et. al. [LLPY97] propose a compact encoding for DBMs which provides a minimal and canonical representation of clock constraints and allows for efficient inclusion checking between constraint systems. They do not consider how this representation may be combined with a compact representation of the rest of the system.

Behrmann et. al. [BLP+99] have recently proposed clock difference diagrams (CDD's) as a data structure for the compact representation of unions of zones. On a variety of case studies, they report space savings of between 46%–99% over their earlier DBM implementation. Difference decision diagrams (DDD's) are a similar data structure developed by Møller et.al. [ML98]. As yet, they are relatively untried in practice, although experimental results of their application in the analysis of a timed version of Milner's cyclic scheduler are very promising [MLAH99]. Other work in this area are the Interval Diagrams of Strehl [Str99], the Region Encoding Diagrams of Wang [Wan00] and the timed polyhedra of Bournez and Maler [BM00].

[2] Numerical Decision Diagrams

5.8 Conclusions and further work

In this chapter, we have introduced an on-the-fly algorithm for reachability analysis of *bCANDLE* systems. The algorithm allows for the analysis of a system model during construction of its simulation graph, without first requiring construction of its equivalent TA. We have also shown how clock activity reduction can be applied on-the-fly. This is essential, in practice, for the analysis of even moderately sized models. In addition, we have proposed the use of MA's for the representation of the state space in the reachability analysis of timed systems. The advantage of this approach is that a compact representation of the discrete state variables can be combined very simply with a DBM representation of clock zones. Experimental results suggest that this leads to significant space reductions, which are achieved in spite of the inclusion of timing information. The impact of MA's on the space requirements of clock zones is less promising and shows no improvement over the use of variable dimension DBM's. For this reason, we expect to see the greatest benefits in the analysis of asynchronous, data-bearing systems, where the value of this approach has already been demonstrated in untimed settings [GGZ95, Gré96]. The CAN-based systems which we consider fall mainly within this class.

Further work includes applying MA state storage to a wider range of examples in order to confirm the findings reported here. In addition, it may be possible to discover more effective partitionings and variable orderings than those considered so far: techniques based on a static analysis of variable dependencies may offer a promising line of attack. It would also be worthwhile to consider combining MA state storage with orthogonal state vector compression techniques such as the collapse method of [Vis96], the tightening of variable ranges of [GdV99] and the compact DBM encoding of [LLPY97]. It is also necessary to compare the performance of MA state storage with CDD's and DDD's. It is clear that a MA gives a more natural encoding of the discrete state variables, however CDD/DDD's are likely to be more effective in the compact representation of clock zones. More substantial work is needed in order to assess the effectiveness of MA state storage in conjunction with partial order reduction. A potential advantage of the MA approach over CDD/DDD's is that a MA state store does not hinder a standard implementation of p.o. reduction, based on depth-first search with tagging of states already on the stack. It is not yet clear how p.o. reduction can be be combined with a fully symbolic use of CDD/DDD's.

In a wider context, we expect that a MA option would be a useful addition to KRONOS and UPPAAL, now that both tools handle system descriptions with discrete variables.

6. *CANDLE*: MODELLING AND ANALYSIS IN PRACTICE

6.1 Introduction

This chapter presents *CANDLE* – a **CAN D**evelopment **L**anguage and **E**nvironment. The purpose of *CANDLE* is to demonstrate

- a programming language for distributed embedded systems whose components communicate using the CAN protocol, and
- a development environment which integrates a variety of tools to support both system implementation and formal analysis.

Our approach is very much influenced by ESTEREL [BG92], a programming language and tool set for the construction and analysis of uni-processor embedded systems. We aim to provide support for the view that *bCANDLE* offers an effective formal basis to support the ESTEREL philosophy of WYVIWYE ('What You Verify Is What You Execute') in the case of CAN-based distributed systems. The emphasis in this chapter is on the construction and analysis of models, rather than on code generation and system implementation, which are mentioned only in connection with model construction.

Work on *CANDLE* continues. Both the language and the development environment are evolving. This chapter provides a snapshot of the current status. The chapter is organised as follows. An informal 'tour' of the language, in the style of the ESTEREL Language Primer [Ber98a], is presented in §6.2 and a simple data modelling language is introduced in §6.3. The translation to *bCANDLE* is discussed in §6.4. Section 6.5 outlines the main features of the development environment which support the construction and analysis of formal models of a *CANDLE* system. A simple example is described in §6.6. Finally, conclusions and related work appear in §6.7.

6.2 A Tour of *CANDLE*

The *CANDLE* language is intended as a simple, high-level programming language for use in the construction of distributed, CAN-based, embedded systems. It can be used to describe the implementation of CAN system designs, which may have been developed and explored at an abstract level using *bCANDLE*. Particular care has been taken to ensure that a formal model of a system can be automatically extracted from its *CANDLE* implementation. This has the benefits of removing the task of model construction from the system developer and ensuring that the model which is analysed is up to date with the current implementation. A *CANDLE* program can be translated automatically into a program written in a *host language*, such as C or Ada, in order to construct a system implementation, or it can be translated automatically into a *bCANDLE* model, and thence to a Prolog or C implementation of the corresponding labelled transition system, for the purposes of simulation and verification. This section provides an informal introduction to *CANDLE*. A complete grammar appears in Appendix C and details of the construction of a formal model are given in §6.4.

6.2.1 Modules

A *CANDLE* program consists of a collection of *modules*. The *CANDLE* module system is modelled on that of ESTEREL. A module has a name and, optionally, a declaration part and a body which is an executable statement. One module is designated as the main program module. Modules can use sub-modules by executing *module instantiation* statements. If module A uses another module B, then A is said to *depend* on B. The module dependency relation is required to be acyclic, i.e. recursive module instantiation is prohibited.

Here is a simple example of a *CANDLE* module:

```
module Flow is
  const
    PERIOD : duration
  type
    flow_reading
  procedure
    ReadSensor(out flow_reading)
  channel
    k : (flow.flow_reading)
  var
    x : flow_reading
  behaviour
    every PERIOD do
      ReadSensor(x);
      snd(k,flow.x)
    end every
end module
```

The name of the module is `Flow` and it implements the behaviour of the flow sensor task described in §3.7. This is a task which periodically reads a flow sensor and broadcasts its value on a communication channel. This behaviour is shown in the module following the keyword `behaviour`. The preceding sections are declarations of constants, types, procedures, channels and variables. These language features are explained below.

6.2.2 Data declarations

As with ESTEREL, *CANDLE* provides only a very limited facility for describing data types and operations, instead it relies on an external data language to provide the necessary definitions. This approach occasionally appears rather cumbersome but is extremely flexible in practice. It provides portability, and more importantly, facilitates the use of different languages for data modelling and implementation. This allows the user to choose an abstract, non-deterministic language, such as Z, for modelling, and a traditional programming language, such as C or Ada, for implementation.

A simple data modelling language, *SDML*, is introduced later in §6.3 in order to illustrate the use of an external data language with *CANDLE* for the purpose of constructing system models. With some additional work, more fully developed modelling languages such as B [Abr96], VDM [Jon90] and Z [Spi88] could be used instead of *SDML*. This would require the reconcilation of different styles of semantic definition, e.g. the denotational style of Z with the operational style of *bCANDLE*. The restriction of *CANDLE* to finite data types should simplify this problem and future work will seek to exploit this benefit.

Data objects in *CANDLE* are either *pre-defined* or *user-defined*. A few primitive data operations are provided in order to ease the expression of some typical idioms: assignment, comparison and so on. All data objects are global to a program. Each data object used within a module must be declared in that module. In the case that a data object is declared in several modules of a multi-module program, it is required that the declarations are *compatible* (see *module instantiation* §6.2.4).

Types and Operators

CANDLE provides the primitive types `unit`, `boolean`, `id` and `duration`.

- The `unit` type contains the single value `uvalue`.
- The `boolean` type contains the constants `true` and **false**. The operations `and`, `or` and `not` are defined.
- The `id` type is the set of *message identifiers*. For any *CANDLE* program, `id` contains the message identifiers occurring in the channel declarations of the program.
- The `duration` type is the set of time units for *CANDLE* programs. There are pre-defined functions `Secs`, `Msecs`, `Usecs` and `Cycles` which can be used to convert to the `duration` type an integer expression representing seconds, milliseconds, microseconds and clock cycles, respectively. For example, the expression `Msecs(30)` denotes the value of type `duration` which is equivalent to 30 milliseconds.

CANDLE allows the use of integer constants `0, 1, -1, 2, -2, ...` and expressions involving the operators `+`, `-`, `*`, `/` and `mod`. However, there is no unbounded, primitive type `integer`. It is assumed that integer expressions evaluate to an element of some user-defined, finite integer subrange.

User-defined types are introduced into a *CANDLE* program simply by declaring their names in a *type declaration*, for example:

```
type flow_reading
```

Several type names can be introduced in a single type declaration, as follows:

```
type
  byte;
  command;
  resource_status
```

As has been mentioned, user-defined types are abstract, the concrete definitions being given in an external data language.

The relational operators `=`, `/=`, `<`, `<=`, `>=`, and `>` can be used with any data type. If they are used, they must be adequately defined in the external data language.

Constants

Constants are introduced by declaring their name and type, as follows:

```
const N : byte
const PERIOD : duration
```

The value of a constant is defined either in the external data language or through module instantiation. There is no explicit constant value definition in *CANDLE*.

Variables

Variables are assignable objects which have a name and a type. Variables are declared with the `var` declaration, as follows:

```
var x : flow_reading
var wl : water_level
```

The variable declarations of a set of program modules give rise to a single global state space. If a variable is declared in two or more modules of a multi-module program then all declarations must be type compatible. *CANDLE* inherits the notion of type compatibility defined by the external data language.

A variable may be modified by assignments, procedure calls and message receptions. It is not possible to assign a value to a variable in its declaration. Each variable must be initialised explicitly by an executable statement before it is used. It is an error if any variable is referenced by distinct behaviour expressions occurring as the arguments of a parallel composition, i.e. concurrently executing processes cannot communicate via shared variables.

Functions and Procedures

Functions and procedures are introduced by declaring their names and the type of their parameters. Parameters must be declared to have one of the modes `in`, `out` or `inout`, so that an appropriate parameter-passing mechanism can be chosen for the host language, for example: call-by-value for `in` parameters and call-by-reference for `out` and `inout` parameters. The following example illustrates:

```
procedure
  ReadSensor(out flow_reading);
  UseResource()
function
  IsFullQueue(queue) : boolean
```

If a parameter mode is not specified explicitly, the mode is assumed to be `in`. So the declaration of `IsFullQueue` above is equivalent to

```
IsFullqueue(in queue):  boolean.
```

All parameters to a function must be `in` parameters. Neither procedures nor functions can have access to variables, other than local variables, except through their parameter lists. It follows that function evaluation in *CANDLE* is side-effect free.

Channels

Channels are the objects through which processes communicate by passing messages. Message passing is by broadcast using an abstracted CAN protocol. A message consists of a message identifier and an optional data value. A channel declaration introduces the name of a channel and optionally a set of priority ordered message templates which defines the messages which can be communicated by the channel. For example,

```
channel
  k : (ok.unit, node.command)
```

declares a channel called `k` which can transmit two kinds of messages: those consisting of the message identifier `ok` and the unit value `uvalue`, and those consisting of the message identifier `node` and any value of the type `command`. The order of the message templates is significant: higher priority messages are declared first. So, for channel `k`, `ok` messages have higher priority than `node` messages.

In a multi-module program, the complete declaration of a channel is derived from the (possibly partial) declarations of the channel in all modules where they occur. Not every declaration of a channel is required to be complete in itself. A channel declaration is complete when the identifier, priority ordering and value type of every message mentioned in the program body can be determined. Multiple channel declarations must be compatible, which means that they must agree on the priority ordering and value type of all messages. For example, the following are compatible declarations of the channel `k` declared above:

```
channel k              -- declare the name only
channel k : (ok.unit)  -- not all message templates declared
channel k : (ok, node) -- message types not yet specified
```

Whereas these declarations are not compatible:

```
channel k : (node.command, ok.unit) -- wrong priority ordering
channel k : (node.unit)             -- incompatible type
```

Exceptions

CANDLE allows exceptions to be declared and used in `trap` and `exit` statements. An exception has a name and can carry a value. An exception is declared like a variable, by giving its name and the type of value carried:

```
exception SensorFailure : unit
exception Alarm : alarm_t
```

As with variables, exceptions form part of the global state space of a program. If an exception is declared in two or more modules of a multi-module program, the declarations must be type compatible.

6.2.3 Expressions

The expression language of *CANDLE* is very simple. It is built from:

- constant values of the predefined types `unit`, `boolean`, `id` and `duration`, together with the integer constants from some finite integer subrange;
- variable identifiers;
- a small number of built-in operators, including: `and`, `or`, `not`, `+`, `-`, `*`, `/`, `mod`, `=`, `/=`, `<`, `<=`, `>=` and `>`;
- the pre-defined functions `Secs`, `Msecs`, `Usecs` and `Cycles`;
- the exception value operator `?`, which returns the value bound to a named exception;
- user-declared function calls.

Here are some examples of *CANDLE* expressions:

```
temperature > maxTemperature
count mod N = 0 or count < 10
Alarm2String(?Alarm)
Usecs(30)
```

CANDLE adopts the type compatibility rules of the external data language.

6.2.4 Statements

`null` and `idle` statements

The simplest *CANDLE* statements are `null` and `idle`. Execution of the `null` statement terminates instantaneously with no effect on the state of the network or data environment. Execution of the `idle` statement similarly leaves the program context unchanged but delays forever without terminating.

Send and Receive statements

Broadcast message transmission is initiated by the `snd` statement, as follows:

```
snd(k, node.req)
snd(k, ok)
```

The first parameter names the communication channel on which the message is to be transmitted. The second parameter consists of a message identifier and, optionally, a data value, which together constitute the message to be transmitted, e.g. `node.req` where `node` is the message identifier and `req` is the data value. In the case that no data value is given, the unit value is assumed, e.g. `snd(k, ok)` is equivalent to `snd(k, ok.uvalue)`. The `snd` statement is non-blocking.

Willingness to receive a broadcast message is indicated by the `rcv` statement, as follows:

```
rcv(k, node.x)
rcv(k, ok)
```

The first parameter names the communication channel from which the message is to be received. The second parameter gives the required message identifier and, optionally, the data variable to which the received data value is to be assigned, e.g. `rcv(k, node.x)` can receive a message having the identifier `node` and will assign the data value of the message to the variable `x`. In the case that a message carries the unit data value, it is not necessary to specify a data variable to receive it, e.g. `rcv(k, ok)` succeeds when an `ok` message is available on channel `k`.

The `rcv` statement is blocking – if there is no suitable message available, the calling process waits.

Elapse statement

The `elapse` statement is used to cause a process to wait for a specified period of time.

```
elapse Secs(5)
elapse Msecs(10)
elapse Cycles(50)
```

The constant expression denoting the extent of the delay is required to be evaluable at compile-time and must be of type `duration`.

It is assumed that a compiler will generate code for the `elapse` statement which, starting from the initiation of its execution, will produce a delay which is as close as possible to the requested value. Construction of the model of the `elapse` statement needs to take into account how the generated code and the run-time environment operate in creating the delay. This is discussed in more detail in §6.4.

Assignment and Procedure Call

The assignment statement has the form

$$x \texttt{ := } e$$

where x is a variable and e is a data expression. The variable and the expression must be type compatible. The bounds on the time taken to execute an assignment statement for a given variable type are determined either by analysis of the code which is generated to perform the assignment, or by an explicit `bounds` declaration in the external data model, as discussed in §6.3.

A procedure call has the form

$$P(e_1, e_2, ..., e_n)$$

where $e_1 \ldots e_n$ are data expressions, whose mode and type are compatible with the corresponding parameters in the declaration of the procedure P. Bounds on the procedure execution time are determined as for the assignment statement.

Sequential and Parallel statements

CANDLE allows statements to be combined both in *sequence* and in *parallel*. The sequencing of behaviours is described by the sequential composition operator "`;`", e.g.

```
ReadSensor(x) ; snd(k,flow.x)
```

where execution of the procedure `ReadSensor` is immediately followed by execution of the communication statement `snd(k,flow.x)`.

In the behaviour

$$s_1 \ ; \ s_2$$

the statement s_1 is started as soon as the sequence is started. If s_1 terminates, then s_2 is started at once. If s_1 does not terminate, then s_2 is never started.

The *parallel* statement is written using the parallel composition operator "`|`", e.g.

```
ReadSensor(x) ; snd(k,flow.x) | rcv(k,flow.y) ; AdjustValve(y)
```

The parallel composition operator has lower precedence than sequential composition. It is only allowed at the top-level of a behaviour.

In the behaviour

$$s_1 \mid s_2$$

the statements s_1 and s_2 are both started as soon as the parallel behaviour is started, and are assumed to execute concurrently. The parallel behaviour terminates when both s_1 and s_2 terminate. Parallel composition in *CANDLE* is asynchronous and communication is restricted to message passing via broadcast channels. In order to guard against interference between the behaviours s_1 and s_2, it is required that the sets of variables and exceptions to which they refer are disjoint.

If statement

The `if` statement is used to allow the execution of a program to depend upon the value of boolean data expressions. The general form of an `if` statement is

```
if e0 then s0
elsif e1 then s1
⋮
elsif en then sn
else s
end if
```

where e_0 to e_n are boolean expressions and s_0 to s_n are statements, as is s. The `elsif` and `else` parts of the statement are optional. The expressions e_0 to e_n are evaluated in sequence. The first *true* expression causes the corresponding statement to be executed. If none of the expressions evaluates to *true*, then the `else` statement s is executed if it is present, otherwise the `if` statement terminates.

Iteration statements

Repetitive behaviours can be described in *CANDLE* using a variety of iteration constructs. The simplest iteration construct is the basic `loop` statement which allows the expression of a behaviour which is executed repeatedly forever. The *named* `loop` statement extends the basic loop by providing a *name* which can be used in an `exit` statement to cause the named loop to be terminated. The `every` statement allows the description of a behaviour which is executed *periodically.*

A *basic* loop statement has the form

```
loop do
    s
end loop
```

where s is a statement. A basic loop executes the statement s repeatedly forever.

Here is an example of the use of a basic loop in implementing the `Valve` process for the flow regulator example of §3.7:

```
module Valve is
  type
    flow_reading
  procedure
    AdjustValve(flow_reading)
  channel
    k : (flow.flow_reading)
  var
    x : flow_reading
  behaviour
    loop do
      rcv(k,flow.x);
      AdjustValve(x)
    end loop
end module
```

The process repeatedly waits to receive a *flow* message and then adjusts a valve accordingly.

A *named* loop statement has the form

```
loop LoopName do
    s
end loop
```

where s is a statement and *LoopName* is an identifier.

In the case of a named loop, an `exit` statement, occurring as part of the statement s, causes the loop to be terminated, e.g.

```
x := 0;
loop Transmit do
  snd(k, value.x);
  x := x + 1;
  if x = 10 then exit Transmit end if
end loop
```

The `Transmit` loop is terminated after ten iterations.

Another form of repetition is introduced in *CANDLE* by the `every` statement, which has the form:

```
every T do
    s
end every
```

where T is a statically evaluable constant expression of type `duration` and s is a statement. The `every` statement causes s to be executed periodically, with execution beginning every T time units. For example,

```
every Msecs(10) do
   ReadSensor(x);
   snd(k, flow.x)
end every
```

causes execution of the statement body to be initiated immediately and to be executed periodically every 10 msecs thereafter.

Select statement

A basic `select` statement allows a choice to be made from several alternative statements, depending on the reception of a message or the elapse of a time delay. It has the general form:

$$
\begin{array}{l}
\texttt{select} \\
\quad :: \;\; \texttt{rcv}(k_1, i_1.x_1) \; ; \; s_1 \\
\quad :: \;\; \texttt{rcv}(k_2, i_2.x_2) \; ; \; s_2 \\
\quad \vdots \\
\quad :: \;\; \texttt{rcv}(k_n, i_n.x_n) \; ; \; s_n \\
\quad :: \;\; \texttt{elapse}\, T \; ; \; s \\
\texttt{end select}
\end{array}
$$

If one of the $\texttt{rcv}(k_j, i_j.x_j)$ statements succeeds, then the program continues by executing the statement s_j. If more than one of the `rcv` statements can succeed simultaneously, then a choice between them is made non-deterministically. If no `rcv` statement can succeed before T time units have elapsed, then statement s is executed.

CANDLE also offers an extended `select` statement, which has the form

$$
\begin{array}{l}
\texttt{select} \\
\quad :: \;\; \texttt{rcv}(k_1, i_1.x_1) \; ; \; s_1 \\
\quad :: \;\; \texttt{rcv}(k_2, i_2.x_2) \; ; \; s_2 \\
\quad \vdots \\
\quad :: \;\; \texttt{rcv}(k_n, i_n.x_n) \; ; \; s_n \\
\quad :: \;\; \texttt{elapse}\, T \; ; \; s \\
\texttt{in} \\
\quad \textit{body} \\
\texttt{end select}
\end{array}
$$

and behaves like a basic select statement, except that the statement *body* is executed while a message reception or timeout is awaited. If a message is received or the time delay elapses before *body* terminates, then the execution of *body* is aborted and execution of the corresponding statement is started. If *body* terminates before a message is received or the time delay elapses then the `select` statement terminates.

The following example illustrates both forms of the `select` statement:

```
select
  :: rcv(k,shutdown); ShutDown(); idle
in
  loop do
    select
      :: rcv(k,pump_on) ; PumpOn()
      :: rcv(k,pump_off) ; PumpOff()
    end select
  end loop
end select
```

The body of the outer `select` statement is a `loop` which repeatedly waits for either a `pump_on` or `pump_off` message and then executes the appropriate procedure. However, if a `shutdown` message is received, the `loop` is aborted, the `ShutDown` procedure is executed and the process idles.

Trap and Exit statements

The `trap` statement can be used to trap exceptions raised in a program block and to define an appropriate behaviour for handling each trapped exception. The `trap` statement has the general form:

$$
\begin{array}{l}
\texttt{trap} \\
\quad \texttt{::} \;\; x_1 \; \texttt{=>} \; s_1 \\
\quad \texttt{::} \;\; x_2 \; \texttt{=>} \; s_2 \\
\quad \vdots \\
\quad \texttt{::} \;\; x_n \; \texttt{=>} \; s_n \\
\texttt{in} \\
\quad \textit{body} \\
\texttt{end trap}
\end{array}
$$

where each x_i is a previously declared exception identifier and each s_i is a statement which acts as the handler for exception x_i. Execution of the `trap` statement begins by executing the statement *body*. An exception can be raised in *body* by using the `exit` statement. If an exception is raised, the execution of *body* is aborted and, if the exception is trapped, execution of the exception handler is started. In the case of a valued exception, the `exit` statement is used to define the value of the exception, e.g.

```
exit Alarm(flowHigh)
```

raises the `Alarm` exception and binds to it the value `flowHigh`. Notice that the value of an exception can be referred to in its handler by using the `?` operator, as in

```
trap
    ⋮
    :: Alarm => if ?Alarm = flowHigh then ... end if
    ⋮
in
    ⋮
    exit Alarm(flowHigh);
    ⋮
end trap
```

It is an error to attempt to refer to the value of an exception outside its handler.

Module Instantiation

A module can be instantiated within another module by using a module instantiation statement. This has the forms

```
M     -- module identifier, no renaming
M[R]  -- module identifier, with renaming
```

where M is the name of a module and R is a list of renamings. The instantiation is syntactically replaced by the body of the module M renamed according to R. A renaming e/I causes all occurrences of the identifier I in M to be replaced with the expression e. This is simple textual replacement; e is not evaluated at this point. The resulting module must be well-formed.

All declarations are global to a *CANDLE* program. Therefore, the declarations of the instantiated module are exported to the parent module. If the parent and child modules both declare objects having the same name, then the declarations must be compatible. Compatibility for constants, variables, procedures and functions is simply type compatibility as defined by the external data language. Compatibility for channels is described in §6.2.2 page 150.

Here is an example of the use of module instantiation, which uses the `Flow` and `Valve` modules declared earlier.

```
module FlowRegulator is
  behaviour
    Flow[Msecs(10)/PERIOD] | Valve[y/x]
end module
```

The details for the expansion of a module instantiation are as given below.

Firstly, a module is called *independent* if it does not contain any module instantiation statements in its body; otherwise it is said to be *dependent*.

- To expand an independent module instantiation $M1[R]$ in a parent module M:

```
module FlowRegulator_E is
  type
    flow_reading
  procedure
    ReadSensor(out flow_reading);
    AdjustValve(flow_reading)
  channel
    k : (flow.flow_reading)
  var
    x : flow_reading;
    y : flow_reading
  behaviour
    every Msecs(10) do
      ReadSensor(x);
      snd(k,flow.x)
    end every
  |
    loop do
      rcv(k,flow.y);
      AdjustValve(y)
    end loop
end module
```

Fig. 6.1: Flow Regulator: Instantiated and Renamed

1. Apply the renaming R to $M1$, giving the renamed module $M1'$.
2. Textually replace the module instantiation statement with the body of the module $M1'$.
3. Merge the declarations of $M1'$ with the declarations of its parent module M.

- To expand a dependent module instantiation $M1[R]$ in a parent module M:

  1. Recursively expand any module instantiations in the body of $M1$.
  2. Expand the remaining independent module instantiation in M, as described above.

The effect of applying these rules in expanding the instantiations in the module `FlowRegulator` is shown in the module `FlowRegulator_E` of Figure 6.1.

6.3 *SDML*: Simple Data Modelling Language

We require an external data language in order to provide complete examples of the modelling of systems using *CANDLE*. It is outside the scope of this thesis to discuss the connection of *CANDLE* to a standard data language such as Z. Instead, we introduce a simple data modelling language, *SDML*, which is an extension of Dijkstra's non-deterministic language of guarded commands [Dij76].

A *SDML* program is just a sequence of type, constant, function and procedure declarations. This section gives an informal introduction to *SDML*. A complete grammar is given in Appendix D.

6.3.1 Types

SDML has the same pre-defined types as *CANDLE*: `unit`, `boolean`, `id` and `duration`. In addition, the following types can be constructed:

- *enumeration* types, which are declared by enclosing between braces a comma-separated list of the values of the type, e.g. `{low, ok, high}`;
- *subrange* types, which have the form *low* `..` *high*, where *low* and *high* are expressions which are evaluable at compile-time and denote values of some ordered type; values of the subrange type are all those of the underlying ordered type from *low* to *high* inclusive, e.g. `0..4` defines the values `0,1,2,3` and `4`;
- *record* types, which are are tuples of named elements, enclosed by the delimiters `{|` and `|}`, e.g.
  `{| numerator : 0..9999; denominator : 0..9999 |}`
  consists of a pair of integers in the range `0..9999`;
- *array* types, which are sequences of values of some previously defined type, indexed by a subrange of some ordered type, e.g. `array 0..4 of boolean`.

A type can be given a name in a type declaration, as follows:

```
type flow_reading is unit
type water_level is {low, ok, high}
type byte is 0..255
type rational is {| numerator   : 0..9999;
                    denominator : 0..9999 |};
     byte_array is array 0..3 of byte
```

Recursive type declarations are not allowed.

In modelling the data of a system, it is usually the case that we abstract from the full set of data values of the underlying implementation and use a smaller set of values which is large enough to preserve the system properties of interest. For example, in the declaration of `flow_reading` above, we have abstracted entirely from the set of flow readings and use the `unit` type instead. However, in order to calculate the communication latency of messages which contain `flow_reading` data, it is necessary to know the size of its representation as implemented. We extend type declarations to allow this information to be included:

```
type flow_reading is unit size Bytes(4)
```

where the `size` clause introduces an expression giving the size of the implemented data representation of the type. The pre-defined functions `Bytes` and `Bits` can be used in `size` expressions.

6.3.2 Constants

Constants are declared using the keyword `const`:

```
const
  req             : command;
  NUMBER_OF_NODES : 0..255 is 10;
  MAX_TEMPERATURE : 0..65535 is 25000
```

where each constant is declared by giving its name, its type and, optionally, its value. The value of a constant is given by an expression following the keyword `is`, as in

```
const NUMBER_OF_NODES : 0..255 is 10.
```

An expression used in a constant definition must be evaluable at compile-time.

6.3.3 Expressions

Expressions in *SDML* are the same as in *CANDLE*, with the following extensions:

- The pre-defined functions `Bytes` and `Bits` are provided for use in `size` declarations.
- The *non-deterministic* expression `any` type*Identifier*, evaluates to any one of the values of the type denoted by type*Identifier*. For example, the expression `any water_level` evaluates to any one of `low`, `ok` or `high`. The use of the `any` expression is restricted to simple assignment, e.g. `wl := any water_level`.
- A *field selector* expression has the form *x.f*, where `x` is a record variable and *f* is a field. For example, if `x` is a record variable whose value is `{| numerator = 1; denominator = 2 |}`, then the value of `x.numerator` is `1` and the value of `x.denominator` is `2`.
- An *array element selector* expression has the form *a*[*i*], where *a* is an array variable and *i* is an expression denoting a value of the index type of *a*. For example, if `a` is an array variable whose value is `[| 0; 2; 4; 8 |]`, then `a[2]` is an expression whose value is `4`, assuming that the index type of `a` is `0..3`.

6.3.4 Functions and Procedures

Function and procedure declarations consist of a *header*, which has the same syntax as in *CANDLE* and a *body* which is written after the keyword `is`:

```
function IsEmptyQueue(q : queue) : boolean is
  bounds Cycles(30) ; Cycles(45)
  begin
    return (q.rear = 0)
  end
```

```
procedure Swap(inout x : byte; inout y : byte) is
  bounds Usecs(100) ; Usecs(125)
  var temp : byte
  begin
    temp := x;
    x := y;
    y := temp
  end
```

The body of the function or procedure consists of a *bounds* declaration, a *local variable* declaration and a *statement*. The bounds declaration allows the user to state lower and upper bounds on the execution time of the function or procedure. In the declaration

```
bounds Cycles(30) ; Cycles(45)
```

the lower (resp. upper) bound is `30` (resp. 45) clock cycles. The expression for each bound must be of type `duration`. It is usual to state bounds in clock cycles and to allow a `duration` value to be calculated automatically when the execution environment is fixed for a particular invocation of the sub-program. However, it is possible to state bounds which are independent of the execution environment, as in the declaration of `Swap`. In the declaration `bounds` t^{lb} `;` t^{ub}, it is required that $t^{\mathsf{lb}} \leq t^{\mathsf{ub}}$. The 'infinite bound' ∞ can be used and is written as `~`, e.g. `bounds Cycles(30); ~`.

A function declaration is required to respect the following constraints:

- the non-deterministic assignment statement is not allowed in a function body, nor in the body of any procedure which is called by a function;
- all function parameters are required to have the mode `in`;
- the only variables which can be referred to in the body of a function are actual parameters and local variables.

6.3.5 Statements

A *SDML* statement is either an atomic statement or a sequential statement. The atomic statements are:

- the `skip` statement which terminates leaving the data state unchanged;
- the assignment statement x `:=` e which causes the value of the expression e to be bound to the variable x;
- the procedure call statement $P(e_1, \ldots e_n)$, where P is the name of a procedure and e_1 to e_n are the actual parameters;
- the return statement `return` e which is used in a function body to indicate that the value of the function is e;

- the non-deterministic `if` statement

$$
\begin{array}{l}
\texttt{if} \\
\quad \texttt{::} \quad e_1 \ \texttt{=>} \ s_1 \\
\quad \texttt{::} \quad e_2 \ \texttt{=>} \ s_2 \\
\quad \vdots \\
\quad \texttt{::} \quad e_n \ \texttt{=>} \ s_n \\
\texttt{fi}
\end{array}
$$

  where each e_i is a boolean expression, called a *guard*, and each s_i is a statement which can be chosen for execution if the associated guard evaluates to *true*. When more than one guard is *true*, the statement to be executed is chosen non-deterministically from among the statements whose guards are *true*. It is required that at least one of the guards in an `if` statement is *true*.

- the non-deterministic `do` statement

$$
\begin{array}{l}
\texttt{do} \\
\quad \texttt{::} \quad e_1 \ \texttt{=>} \ s_1 \\
\quad \texttt{::} \quad e_2 \ \texttt{=>} \ s_2 \\
\quad \vdots \\
\quad \texttt{::} \quad e_n \ \texttt{=>} \ s_n \\
\texttt{od}
\end{array}
$$

  whose branches are as for the `if` statement. If some guard evaluates to *true*, a statement is chosen for execution and the `do` statement is repeated. The `do` statement terminates when no guard evaluates to *true*. The user is required to establish the termination of every `do` statement in a *SDML* program.

In a sequential statement s_1 `;` s_2, the statement s_1 is executed and, when the execution of s_1 terminates, execution of s_2 begins.

6.3.6 Semantics

SDML is a block-structured, statically-scoped, sequential programming language. It introduces a few familiar mechanisms for declaring types, constants, functions and procedures. Statements are essentially as in Dijkstra's guarded command language [Dij76]. We assume the existence of a semantic function which gives the meaning of *SDML* statements. This function is required later in constructing a *bCANDLE* model from a *CANDLE* program where *SDML* is used as the external data language.

Let *Statement* be the set of *SDML* statements. For a *SDML* program, let *Var* be the set of data variables and V be the set of data values. Let $\mathit{Valuation} \mathrel{\widehat{=}} \mathit{Var} \rightarrow V$ be the set of valuations. Then, the semantic function

$$\mathcal{S} : \mathit{Statement} \rightarrow \mathit{Valuation} \rightarrow 2^{\mathit{Valuation}}$$

gives the meaning of *SDML* statements, where $\mathcal{S}\,[\![s]\!]\mathsf{val}$ denotes the set of valuations which are possible results of executing the statement s under the valuation val.

Notice that because *SDML* is a non-deterministic language, a statement maps a valuation to a *set* of result valuations. Nevertheless, the definition of the semantic function is quite straightforward; the interested reader is referred to standard texts such as Schmidt [Sch86] or Winskel [Win93] for further details.

6.4 Constructing a Formal Model

This section describes how a *CANDLE* program can be translated into *bCANDLE*, so that its behaviour can be simulated or verified. The construction of a *bCANDLE* model, which conservatively approximates the implemented system, depends not only on the *CANDLE* program but also on features of the code generator and the execution environment. It is outside the scope of this thesis to discuss these aspects fully. The intention here is to provide a general framework for the translation, which can be adapted to accommodate particular requirements.

It is assumed, without loss of generality, that a *bCANDLE* model is constructed from a single *CANDLE* module of the form:

```
module moduleName is
    type typeDecl1; ...; typeDecln
    const constantDecl1; ...; constantDecln
    var variableDecl1; ...; variableDecln
    function functionDecl1; ...; functionDecln
    procedure procedureDecl1; ...; procedureDecln
    channel channelDecl1; ...; channelDecln
    exception exceptionDecl1; ...; exceptionDecln
    behaviour statement
end module
```

and a single *SDML* module of the form:

```
data moduleName is
    type typeDecl1; ...; typeDecln
    const constantDecl1; ...; constantDecln
    function functionDecl1; ...; functionDecln
    procedure procedureDecl1; ...; procedureDecln
end data
```

That is to say, module instantiation statements are expanded, and declarations are collected, to give a well-formed, stand-alone *CANDLE* module; and the external data definitions are presented as a single *SDML* module.

Recall that a *bCANDLE* model is a tuple (P, N, D), where P is a process term, N is a network and D is a data environment (§3.6). In the rest of this section, we show how each component of the model can be constructed from a

CANDLE program. First, we consider how the data environment and network model are constructed from *CANDLE* declarations; then, how a process term is constructed from a `behaviour` section.

6.4.1 Declarations

Data

A *bCANDLE* data environment is a tuple $(\mathsf{type}, \mathsf{operation}, \mathsf{predicate}, \mathsf{val})$ (§3.3). This section shows how a *CANDLE* program defines a *bCANDLE* data environment.

A valuation $\mathsf{val} : \mathit{Var} \to V$ is a mapping from variables to values. The set Var of variables is defined by the *CANDLE* `var` and `exception` declaration sections. There is one *bCANDLE* variable for each declared *CANDLE* variable and exception. In addition, Var includes a number of *system variables* which are not referred to in the *CANDLE* program but are used to hold the values of expressions occurring in the `behaviour` section. In constructing the formal model, we assume that there is a unique system variable for each program expression. In practice, a smaller number of variables are used and expressions are assigned to them according to principles which ensure that conflict is avoided. The type of a variable is given either directly by its declaration or, in the case of a system variable, can be inferred from the type of the expression whose value is bound to it. Furthermore, each *SDML* type expression clearly denotes a finite set of values. So each variable $x \in \mathit{Var}$ is associated with a finite set V_x of values, which is given by the type of x. The set V of all program data values is then given by

$$V \mathrel{\widehat{=}} \bigcup_{x \in \mathit{Var}} V_x \cup \{\bot\},$$

where $\bot$ represents the distinguished "undefined" value. The function $\mathsf{type} : \mathit{Var} \to 2^V$ is defined simply by

$$\mathsf{type}(x) \mathrel{\widehat{=}} V_x$$

for all $x \in \mathit{Var}$. A valuation $\mathsf{val} : \mathit{Var} \to V$ maps each variable x to some value v, where either $v \in \mathsf{type}(x)$ or $v = \bot$. For any *CANDLE* program, the initial valuation maps every variable to $\bot$.

The operation symbols and predicate symbols of the *bCANDLE* model are determined during the translation of the `behaviour` section, as are their interpretations. Consideration of the details is deferred to §6.4.2.

Network

A *bCANDLE* network is a mapping from channel identifiers to channels (§3.4), where a channel is defined by its *static* and *dynamic* attributes. This section shows how a *CANDLE* channel declaration section

$$\texttt{channel}\ \mathit{channelDecl}_1;\ \ldots;\ \mathit{channelDecl}_n$$

defines a *bCANDLE* network.

Each channel declared in a *CANDLE* channel declaration is modelled by its own distinct *bCANDLE* channel, whose attributes are constructed as follows:

Static attributes In constructing the static attributes of a channel, we need to identify the message set M, the priority ordering $\prec$ and the transmission latency function δ. The message set and priority ordering are constructed from the *CANDLE* declaration of the channel and the declaration of the message data types, e.g. the declarations

```
type command is (req, rel)

channel k : (ok.unit, node.command)
```

define a message set

$$M = \{\texttt{ok.uvalue}, \texttt{node.req}, \texttt{node.rel}\}$$

and a priority ordering

$$\texttt{ok} \prec \texttt{node}.$$

The construction of the transmission latency function δ depends not only on the *CANDLE* channel and data declarations, but also on the characteristics of the physical communication links to which the channels are mapped by the system architecture. For example, assume that `k` is mapped to a CAN bus operating at $5 \times 10^5\, bit/s$, in which the acceptance test coincides with the leading edge of bit ACK0 (see Figure 1.2). Assume also that 1 unit of duration $= 1\,\mu$secs. Then, the transmission latency function is as follows:

δ	units of `duration`		
	`ok.uvalue`	`node.req`	`node.rel`
δ^{lb}	70	86	86
δ^{ub}	86	106	106
δ^{lB}	24	24	24
δ^{uB}	24	24	24

Here, the calculation of δ assumes CAN packets of 0 data bytes for `ok` messages and 1 data byte for `node` messages. As an illustration of the calculation, consider $\delta^{\mathsf{ub}}(\texttt{node.req})$. In a CAN packet with 1 data byte, there are 43 bits from SOF up to, but not including, ACK0. A stuff bit is inserted after every 5 consecutive transmitted bits of the same value. Bit stuffing occurs from SOF up to, but not including, the CRC delimiter. The pattern of transmitted bits containing the maximum number of stuff bits is of the form

$$00000[1]1111[0]0000[1]1111[0]\ldots$$

where the inserted stuff bits are shown in brackets. So, in the example, there are at most $\lfloor 41/4 \rfloor = 10$ stuff bits, and thus, at most $43 + 10 = 53$

transmitted bits, before the acceptance test of a `node.req` packet. At a data rate of $5 \times 10^5\, bit/s$, 53 bits are transmitted in 106 μsecs. The other values for δ are calculated similarly. Notice that it is only coincidence that $\delta^{\mathsf{ub}}(\texttt{ok.uvalue}) = \delta^{\mathsf{lb}}(\texttt{node.req})$. It just happens that the maximum number of stuff bits for a CAN packet containing 0 data bytes is 8 bits, just the same as the number of extra bits in a CAN packet containing 1 data byte and no stuff bits.

Dynamic attributes The dynamic attributes of a channel are its status and its pending message queue. The initial status of a channel is defined to be FREE and the initial pending message queue is empty.

6.4.2 Behaviour

A *bCANDLE* process term is constructed from the `behaviour` section of a *CANDLE* program as described below. The translation of a *CANDLE* behaviour depends upon the semantic function $\mathcal{S}$ which gives the meaning of *SDML* statements (§6.3.6). This is required to define the results of executing an assignment statement or procedure call. A semantic function is similarly required to give a meaning to *CANDLE* expressions. Let *Expression* denote the set of *CANDLE* expressions. For a *CANDLE* program, let *Var* be the set of data variables and V the set of data values. Let $Valuation \mathrel{\widehat{=}} Var \rightarrow V$ be the set of valuations. Then,

$$\mathcal{E} : Expression \rightarrow Valuation \rightarrow V$$

is the semantic function which gives a meaning to *CANDLE* expressions, and $\mathcal{E}\,[\![e]\!]\mathsf{val}$ denotes the value of the expression e under the valuation val.

Now, the translation from a *CANDLE* program to a *bCANDLE* model can be given inductively, as follows.

Null and Idle statements

Both `null` and `idle` leave the data state unchanged. The difference is that `null` terminates immediately whereas `idle` never terminates. They have direct counterparts in *bCANDLE*:

- $[\![\texttt{null}]\!] \mathrel{\widehat{=}} \mathrm{null}$
- $[\![\texttt{idle}]\!] \mathrel{\widehat{=}} \mathrm{idle}$

Send and Receive statements

Each communication, $\texttt{snd}(k, i.e)$ and $\texttt{rcv}(k, i.x)$, requires some computation time both before and after it, not only to evaluate the expression e in the case of `snd`, but perhaps also to configure a communication controller or modify the process status; the particular details depend upon the execution environment, which must be analysed in order to calculate the required execution bounds.

Let *pre_snd* (resp. *post_snd*) denote the bounds on the execution time needed before (resp. after) the completion of the `snd` operation. Let *pre_rcv* and *post_rcv* denote the corresponding bounds for the `rcv` operation. Then, the `snd` and `rcv` operations are modelled as follows:

- $[\![\texttt{snd}(k, i.e)]\!] \mathrel{\widehat{=}} [\omega : \textit{pre\_snd}] \; ; \; k!i.x \; ; \; [\textit{post\_snd}]$,
  where x is a system variable allocated to hold the value of the expression e, and ω is a new operation symbol defined by:

  $$\begin{aligned}&\mathsf{operation}(\omega) \mathrel{\widehat{=}}\\ &\qquad \{(\mathsf{val}, \mathsf{val}') \in \textit{Valuation} \times \textit{Valuation} \mid \mathsf{val}' = \mathsf{val}[x := \mathcal{E}\,[\![e]\!]\mathsf{val}]\}\end{aligned}$$

- $[\![\texttt{rcv}(k, i.x)]\!] \mathrel{\widehat{=}} [\textit{pre\_rcv}] \; ; \; k?i.x \; ; \; [\textit{post\_rcv}]$

Elapse statement

The implementation of the `elapse` statement requires access to a timer service provided by the execution environment. It is assumed that some time-consuming operations are required both before and after the requested delay. What operations are needed, and how much time they consume, is determined by the particular implementation, and may include: calling a timer service routine, configuring a hardware timer, rescheduling a process after a delay expiry, and so on. In addition, the duration of the implemented delay may only approximate the requested delay. The model of the `elapse` statement seeks to account for such implementation details.

Let *pre_timer* (resp. *post_timer*) denote the bounds on the computation time required before (resp. after) a request to use a timer service. Let *approx* T denote the bounds on the actual delay delivered by a request for a delay of T time units. Then, the `elapse` statement is modelled as follows:

- $[\![\texttt{elapse}(T)]\!] \mathrel{\widehat{=}} [\textit{pre\_timer}] \; ; \; [\textit{approx} \;\; T] \; ; \; [\textit{post\_timer}]$

Assignment and Procedure Call

An assignment statement of the form x `:=` e, where x is a variable whose type is denoted by *type_id*, is treated as syntactic sugar for a procedure call:

$\texttt{assign\_}\textit{type\_id}(x, e)$

which is assumed to have the declaration

$\texttt{procedure assign\_}\textit{type\_id}(\texttt{out}\ \textit{type\_id}, \texttt{in}\ \textit{type\_id})$

The translation of an assignment statement is then given by the translation of its corresponding procedure call, as explained below.

A procedure call has the form:

$P(e_1, \ldots, e_n)$

where P is the name of the procedure and each e_i is an expression denoting an actual parameter of P. It is assumed that all parameters are evaluated before the procedure executes. Let t_i^{lb} (resp. t_i^{ub}) denote the lower bound (resp. upper bound) on the time required to complete the evaluation of e_i. Let $t^{\mathsf{lb}}(P)$ (resp. $t^{\mathsf{ub}}(P)$) denote the lower bound (resp. upper bound) on the time required to execute the procedure P once all its parameters have been evaluated. Then, the translation of $P(e_1, \ldots, e_n)$ is given by:

- $[\![P(e_1, \ldots, e_n)]\!] \mathrel{\widehat{=}} [\omega : t^{\mathsf{lb}}, t^{\mathsf{ub}}]$,
  in which ω is a new operation symbol defined by:

  $$\begin{aligned}&\mathsf{operation}(\omega) \mathrel{\widehat{=}}\\ &\qquad \{(\mathsf{val}, \mathsf{val}') \in \mathit{Valuation} \times \mathit{Valuation} \mid \mathsf{val}' \in \mathcal{S}\,[\![P(e_1, \ldots, e_n)]\!]\mathsf{val}\}\end{aligned}$$

  where $t^{\mathsf{lb}} \mathrel{\widehat{=}} t^{\mathsf{lb}}(P) + \Sigma_{i=1}^{n} t_i^{\mathsf{lb}}$ and $t^{\mathsf{ub}} \mathrel{\widehat{=}} t^{\mathsf{ub}}(P) + \Sigma_{i=1}^{n} t_i^{\mathsf{ub}}$.

If statement

The `if` statement has the form:

$$\texttt{if}\ e_1 \mapsto s_1, \ldots, e_{n-1} \mapsto s_{n-1},\ \texttt{true}\ \mapsto s_n\ \texttt{end if}$$

where each e_i is a boolean expression and each s_i is a statement. The implementation of the `if` statement evaluates each expression e_i in turn and executes the corresponding statement s_i of the first expression whose value is *true*.

- $[\![\texttt{if}\ e_1 \mapsto s_1, \ldots, e_n \mapsto s_n\ \texttt{end if}\ ]\!] \mathrel{\widehat{=}}$
  $[t_1^{\mathsf{lb}}, t_1^{\mathsf{ub}}]\,;\,(\gamma_1 \rightarrow [\![s_1]\!] + \overline{\gamma_1} \rightarrow [\![\texttt{if}\ e_2 \mapsto s_2, \ldots, e_n \mapsto s_n\ \texttt{end if}]\!])$,
  where t_1^{lb} (resp. t_1^{ub}) denotes the lower bound (resp. upper bound) on the time required to complete the evaluation of e_1 and, for $1 \leq i \leq n$, γ_i and $\overline{\gamma_i}$ are new predicate symbols defined by:

  $$\mathsf{predicate}(\gamma_i) \mathrel{\widehat{=}} \{\mathsf{val} \in \mathit{Valuation} \mid \mathcal{E}\,[\![e_i]\!]\mathsf{val} = \mathit{true}\},$$

  and

  $$\mathsf{predicate}(\overline{\gamma_i}) \mathrel{\widehat{=}} \{\mathsf{val} \in \mathit{Valuation} \mid \mathcal{E}\,[\![e_i]\!]\mathsf{val} = \mathit{false}\}.$$

- $[\![\texttt{if true}\ \mapsto s\ \texttt{end if}]\!] \mathrel{\widehat{=}} [\![s]\!]$.

Select statement

Consider a `select` statement of the form:

$$\texttt{select ::}\ \ g_1\,;\,s_1 \ldots\ \texttt{::}\ \ g_n\,;\,s_n\ \texttt{end select}$$

where each g_i is either a `rcv` statement or an `elapse` statement. The statement g_i acts as a guard to entry of the ith alternative in the `select` statement. Notice that the translation of an individual guard statement g has the form

$$[\![g]\!] = [\mathit{pre\_g}]\,;\,\beta\,;\,[\mathit{post\_g}],$$

where, $[\mathit{pre\_g}]$ is either $[\mathit{pre\_rcv}]$ or $[\mathit{pre\_timer}]$, β is either $k?i.x$ or $[\mathit{approx}\ T]$, and $[\mathit{post\_g}]$ is either $[\mathit{post\_rcv}]$ or $[\mathit{post\_timer}]$. However, there is a variety of different ways in which a set of guards can be implemented when used in a `select` statement. Clearly, some computation is required to configure at least one communication request or delay before one of the `select` guards can be executed. However, when several communications and delays must be configured, an implementation has several degrees of freedom, including:

- the order in which the configurations are completed;
- whether all configurations must be completed before one of the `select` alternatives can begin execution.

The translation given below assumes that before a `select` alternative can be chosen:

- at least one configuration has been completed, giving a minimum set up time

  $$t^{\mathsf{lb}} = \mathsf{min}\{t_i^{\mathsf{lb}} \mid 1 \leq i \leq n\},$$

- possibly, all configurations have been completed, giving a maximum set up time

  $$t^{\mathsf{ub}} = \Sigma_{i=1}^{n} t_i^{\mathsf{ub}},$$

where $[\![g_i]\!] = [t_i^{\mathsf{lb}}, t_i^{\mathsf{ub}}] \mathbin{;} \beta_i \mathbin{;} [post\_g_i]$;

The translation of the `select` statement is then:

- $[\![\texttt{select ::}\ \ g_1 \mathbin{;} s_1 \ldots\ \texttt{::}\ \ g_n \mathbin{;} s_n\ \texttt{end select}]\!] \mathrel{\widehat{=}}$
  $[t^{\mathsf{lb}}, t^{\mathsf{ub}}] \mathbin{;}$
  $(\beta_1 \mathbin{;} [post\_g_1] \mathbin{;} [\![s_1]\!] + \beta_2 \mathbin{;} [post\_g_2] \mathbin{;} [\![s_2]\!] + \cdots + \beta_n \mathbin{;} [post\_g_n] \mathbin{;} [\![s_n]\!])$,
  where $[\![g_i]\!] = [t_i^{\mathsf{lb}}, t_i^{\mathsf{ub}}] \mathbin{;} \beta_i \mathbin{;} [post\_g_i]$, $t^{\mathsf{lb}} = \mathsf{min}\{t_i^{\mathsf{lb}} \mid 1 \leq i \leq n\}$ and $t^{\mathsf{ub}} = \Sigma_{i=1}^{n} t_i^{\mathsf{ub}}$.

Of course, it is possible to modify this model to accommodate more elaborate assumptions about the implementation, and this may lead to a tightening of the bounds which are derived using the weak assumptions given here.

Now consider an extended `select` statement of the form:

$$\texttt{select ::}\ \ g_1 \mathbin{;} s_1 \ldots\ \texttt{::}\ \ g_n \mathbin{;} s_n\ \texttt{in}\ s\ \texttt{end select}$$

It is translated in a similar way. The difference is that execution of the statement s is started immediately on entry to the extended `select` statement and continues until termination or until interrupted by the execution of one of the guards g_i. This gives the following translation:

- $[\![\texttt{select ::}\ \ g_1 \mathbin{;} s_1 \ldots\ \texttt{::}\ \ g_n \mathbin{;} s_n\ \texttt{in}\ s\ \texttt{end select}]\!] \mathrel{\widehat{=}}$
  $[t^{\mathsf{lb}}, t^{\mathsf{ub}}] \mathbin{;}$
  $([\![s]\!]\, [\!> \beta_1 \mathbin{;} [post\_g_1] \mathbin{;} [\![s_1]\!] + \beta_2 \mathbin{;} [post\_g_2] \mathbin{;} [\![s_2]\!] + \cdots + \beta_n \mathbin{;} [post\_g_n] \mathbin{;} [\![s_n]\!])$,
  where $[\![g_i]\!] = [t_i^{\mathsf{lb}}, t_i^{\mathsf{ub}}] \mathbin{;} \beta_i \mathbin{;} [post\_g_i]$, $t^{\mathsf{lb}} = \mathsf{min}\{t_i^{\mathsf{lb}} \mid 1 \leq i \leq n\}$ and $t^{\mathsf{ub}} = \Sigma_{i=1}^{n} t_i^{\mathsf{ub}}$.

Trap and Exit statements

The `trap` statement has the form:

$$\texttt{trap ::}\ \ x_1\ \texttt{=>}\ s_1 \ldots \texttt{::}\ \ x_n\ \texttt{=>}\ s_n\ \texttt{in}\ s\ \texttt{end trap}$$

where each x_i is an exception identifier and each s_i is a statement.

There are several possible implementations of the `trap` statement. In constructing the corresponding *bCANDLE* model, it is assumed that an exception is represented by a record variable of type
$\texttt{\{|raised : boolean; value :}\ \textit{sometype}\,\texttt{|\}}$,
where for an exception x, x.`raised` is assigned the value `true` when the exception x is raised, and otherwise has the value `false`. x.`value` holds the value assigned to x when it was last raised, and can be referred to in expressions using the notation `?`x.
The translation of the `trap` statement is then given by

- $[\![\texttt{trap ::}\ \ x_1\ \texttt{=>}\ s_1 \ldots \texttt{::}\ \ x_n\ \texttt{=>}\ s_n\ \texttt{in}\ s\ \texttt{end trap}]\!] \mathrel{\widehat{=}}$
  $[\![s]\!]\ [>(\gamma_1 \to [\omega_1 : t^{\mathsf{lb}}, t^{\mathsf{ub}}]\ ;\ [\![s_1]\!]$
  $\quad +\gamma_2 \to [\omega_2 : t^{\mathsf{lb}}, t^{\mathsf{ub}}]\ ;\ [\![s_2]\!]$
  $\quad + \cdots + \gamma_n \to [\omega_n : t^{\mathsf{lb}}, t^{\mathsf{ub}}]\ ;\ [\![s_n]\!])$,

where each γ_i is a new predicate symbol which is true just when the corresponding variable x_i.`raised` is `true`, i.e.

$$\mathsf{predicate}(\gamma_i) \mathrel{\widehat{=}} \{\mathsf{val} \in \textit{Valuation} \mid \mathsf{val}(x_i.\texttt{raised}) = \texttt{true}\}.$$

Each ω_i is a new operation symbol which simply resets x_i.`raised`, i.e.

$$\begin{aligned}&\mathsf{operation}(\omega_i) \mathrel{\widehat{=}}\\ &\quad \{(\mathsf{val}, \mathsf{val}') \in \textit{Valuation} \times \textit{Valuation} \mid \mathsf{val}' = \mathsf{val}[x_i.\texttt{raised} := \texttt{false}]\}\end{aligned}$$

and t^{lb} (resp. t^{ub}) gives the lower bound (resp. upper bound) on the time required to clear an exception and transfer control to its handler.

The `exit` statement has the form `exit` $x(e)$, where x is an exception identifier and e is an expression denoting the value to be associated with x. The translation of the `exit` statement is defined simply to set x.`raised` and bind the value of e to x.`value`:

- $[\![\texttt{exit}\ x(e)]\!] \mathrel{\widehat{=}} [\omega : t^{\mathsf{lb}}, t^{\mathsf{ub}}]\ ;\ \text{idle}$,
  where ω is a new operation symbol defined by

  $$\begin{aligned}&\mathsf{operation}(\omega) \mathrel{\widehat{=}}\\ &\quad \{(\mathsf{val}, \mathsf{val}') \in \textit{Valuation} \times \textit{Valuation} \mid \mathsf{val}' = \mathsf{val}[x.\texttt{raised} := \texttt{true},\\ &\qquad\qquad x.\texttt{value} := \mathcal{E}\,[\![e]\!]\mathsf{val}]\},\end{aligned}$$

  and t^{lb} (resp. t^{ub}) gives the lower bound (resp. upper bound) on the time required to evaluate e and raise the exception.

Loop statement

The basic `loop` statement has the form:

`loop do` s `end loop`

where s is a statement. It is translated simply using a recursive *bCANDLE* process:

- $[\![$`loop do` s `end loop`$]\!] \mathrel{\widehat{=}} \mathrm{rec}\, LOOP.\, [\![s]\!]\, ;\, LOOP$,
  where $LOOP$ is a new process variable.

The named `loop` statement has the form:

`loop` *loopName* `do` s `end loop`

where *loopName* is the name of the loop and s is a statement. It is treated as syntactic sugar for a `trap` statement which encloses a basic `loop` statement, as follows:

- $[\![$`loop` *loopName* `do` s `end loop` $]\!] \mathrel{\widehat{=}}$
  $[\![$`trap ::` *x_loopName* `=> null in loop do` s `end loop end trap`$]\!]$,
  where *x_loopName* is a new exception of type `unit`.

Every statement

The `every`T statement is just syntactic sugar for a loop which executes its body every T time units. It is translated as follows:

- $[\![$`every` T `do` s `end every`$]\!] \mathrel{\widehat{=}}$
  $[\![$`loop do select :: elapse` T `in` s `; idle end select end loop`$]\!]$

Notice that execution of the statement s `; idle` begins immediately on entering the `every` statement. It is assumed that s terminates and `idle` is started before the elapse of T time units. After T time units, `idle` is interrupted and the `loop` repeats.

Sequential and Parallel Composition

The sequential and parallel composition statements of *CANDLE* have direct counterparts in *bCANDLE* and their translation is simple:

- $[\![$`S1 ; S2`$]\!] \mathrel{\widehat{=}} [\![$`S1`$]\!]\, ;\, [\![$`S2`$]\!]$
- $[\![$`S1 | S2`$]\!] \mathrel{\widehat{=}} [\![$`S1`$]\!] \mid [\![$`S2`$]\!]$

6.4.3 An example

The flow regulator example is used to illustrate the translation from *CANDLE* to *bCANDLE*. The *CANDLE* program for the example is reviewed in Figure 6.2. In the following, we consider how each component of the *bCANDLE* model (P, N, D) is derived from the *CANDLE* program.

```
module FlowRegulator_E is
  type
    flow_reading
  procedure
    ReadSensor(out flow_reading);
    AdjustValve(flow_reading)
  channel
    k : (flow.flow_reading)
  var
    x : flow_reading;
    y : flow_reading
  behaviour
    every Msecs(10) do
      ReadSensor(x);
      snd(k,flow.x)
    end every
  |
    loop do
      rcv(k,flow.y);
      AdjustValve(y)
    end loop
end module

data FlowRegulator_E is
  type flow_reading is unit size Bytes(1)

  procedure ReadSensor(out r : flow_reading) is
    bounds Usecs(85)  ; Usecs(90)
    begin
      r := any flow_reading
    end

  procedure AdjustValve(in r : flow_reading) is
    bounds Usecs(200) ; Usecs(300)
end data
```

Fig. 6.2: Flow Regulator in *CANDLE*

Data Environment

The data environment $D = (\mathsf{type}, \mathsf{operation}\,, \mathsf{predicate}, \mathsf{val})$ is constructed as follows.

- There are two program variables `x` and `y`, each of type `flow_reading`. The data module declares `flow_reading` to be a synonym for the `unit` type. So we have

  $$\mathsf{type}(\mathtt{x}) = \mathsf{type}(\mathtt{y}) = \{\mathtt{uvalue}\}$$

- There are two procedure calls in the `behaviour` section of the *CANDLE* module: `ReadSensor(x)` and `AdjustValve(y)`. So, the set of operation symbols is

  $$\Omega = \{ReadSensor_x, AdjustValve_y\},$$

  where the definition of $ReadSensor_x$ is derived from its data module declaration and gives the effect of applying the operation in any data environment:

$$\begin{array}{l} \mathsf{operation}(\mathit{ReadSensor}_x) \mathrel{\widehat{=}} \\ \quad \{ \\ \qquad \{x \mapsto \bot, y \mapsto \bot\} \mapsto \{x \mapsto \mathtt{uvalue}, y \mapsto \bot\}, \\ \qquad \{x \mapsto \bot, y \mapsto \mathtt{uvalue}\} \mapsto \{x \mapsto \mathtt{uvalue}, y \mapsto \mathtt{uvalue}\}, \\ \qquad \{x \mapsto \mathtt{uvalue}, y \mapsto \bot\} \mapsto \{x \mapsto \mathtt{uvalue}, y \mapsto \bot\}, \\ \qquad \{x \mapsto \mathtt{uvalue}, y \mapsto \mathtt{uvalue}\} \mapsto \{x \mapsto \mathtt{uvalue}, y \mapsto \mathtt{uvalue}\} \\ \quad \} \end{array}$$

$\mathsf{operation}(\mathit{AdjustValve}_y)$ is defined similarly (with the roles of x and y reversed).

- The set Γ of predicate symbols is empty, so

$$\mathsf{predicate} \mathrel{\widehat{=}} \emptyset.$$

- Finally the initial valuation maps each variable to the undefined value

$$\mathsf{val} \mathrel{\widehat{=}} \{x \mapsto \bot, y \mapsto \bot\}.$$

Network

In constructing the static attributes of the network, we need to identify, for each channel, the message set M, the priority ordering $\prec$ and the transmission latency function δ. The message set is constructed from the declarations of the channel and the message data types. In the example, the declaration of channel `k` comprises a single message template `flow.flow_reading`, where `flow_reading` is a synonym for the `unit` data type. So the message set M for `k` is the singleton {`flow.uvalue`}. Since there is only a single message in M, the priority relation $\prec$ is just the empty set $\emptyset$. In order to construct the transmission latency function δ for the channel `k`, it is necessary to know some details of the physical channel which implements it. Let us assume as before that `k` is implemented by a CAN bus operating at $5 \times 10^5\, bit/s$. Then, the transmission latency function is as follows:

δ	units of `duration`
	`flow.uvalue`
δ^{lb}	70
δ^{ub}	86
δ^{lB}	24
δ^{uB}	24

All other assumptions are as in §6.4.1.

Behaviour

The process term P, modelling the system behaviour, is derived from the `behaviour` section of the *CANDLE* program. In our example, this comprises the parallel composition of two processes: `every Msecs(10)` ... and `loop do rcv(k, flow.y)` .... We illustrate by considering the translation

```
⟦every Msecs(10) do
    ReadSensor(x);
    snd(k, flow.x)
end every⟧
```

In the first translation step, the `every` statement is unpacked, giving

```
⟦loop do select ::  elapse Msecs(10) in
    ReadSensor(x); snd(k,flow.x) ; idle end loop⟧
```

Next, the `loop` statement is translated into a recursion

rec $LOOP.$⟦`select ::  elapse Msecs(10) in`
`ReadSensor(x); snd(k, flow.x); idle`⟧ ; $LOOP$

The translation of the `select` statement depends upon the translation of `elapse Msecs(10)`, which is given by $[pre\_timer]$; $[approx\ 10000]$; $[post\_timer]$. As before, we assume that 1 unit of `duration` is equivalent to 1 μsec. We now have

rec $LOOP.[pre\_timer]$;
(⟦`ReadSensor(x); snd(k, flow.x); idle`⟧
$[>[approx\ 10000]\ ;\ [post\_timer])\ ;\ LOOP$

The final steps translate the remaining simple statements, giving the result

$$\text{rec } LOOP.[pre\_timer];$$
$$([ReadSensor_x : 85, 90]\ ;\ [pre\_snd]\ ;\ k!flow.x\ ;\ [post\_snd]\ ;\ \text{idle}$$
$$[>[approx\ 10000]\ ;\ [post\_timer])\ ;\ LOOP$$

where all that remains is to 'plug in' the bounds denoted by pre_timer, $approx$ 10000, $post\_timer$, pre_snd and $post\_snd$.

The remaining process is translated similarly, giving

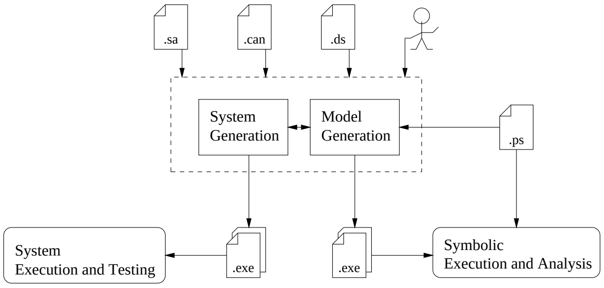
$$\text{rec } LOOP.[pre\_rcv]\ ;\ k?flow.y\ ;\ [post\_rcv]\ ;\ [AdjustValve_y : 200, 300]\ ;\ LOOP$$

6.5 The *CANDLE* Development Environment

6.5.1 Overview

The development of a high-integrity embedded system requires the use of a wide variety of software tools, including: text editors, compilers, simulators, model-checkers, theorem-provers and test case generators. A computer-aided development environment is required which

- is *open* and *extensible*, making it possible to combine different tools to provide implementation and validation functions as required;



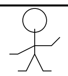
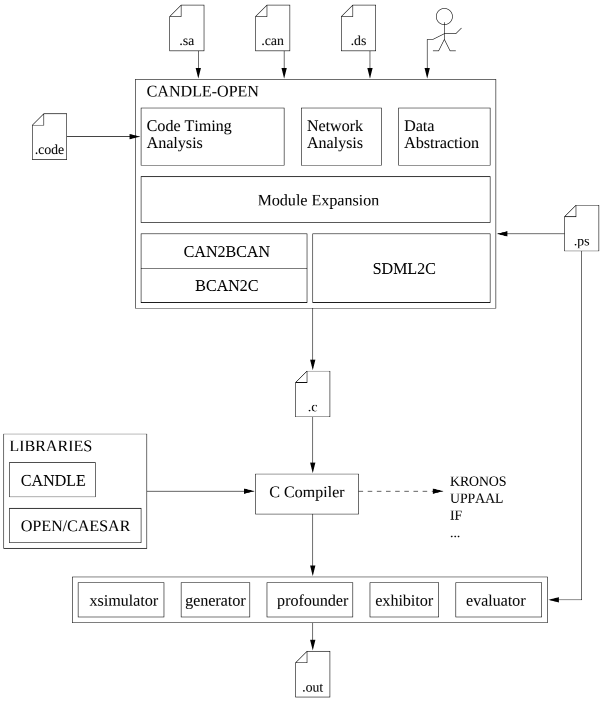
	User commands for all tools are entered using a graphical user interface to the development environment which checks consistency.
`.ds`	Specification files for the data state and sequential operations of each system process. Model-based specification languages such as B, Z or VDM can be used. Sequential code is developed from specifications using a standard methodology, e.g. refinement. Abstract data model is extracted from the same specifications for system verification.
`.can`	*CANDLE* program modules: contain a description of the dynamic behaviour of processes including communication and synchronisation. Declare broadcast channels, including message identifiers and their priorities.
`.sa`	System architecture files: map processes to processors, communication channels to CAN buses, etc.; describe the properties of system components, e.g. processors, CAN buses and hardware timers in order to allow the prediction of timing properties.
`.ps`	Property specification file: a specification of system properties using a logic such as TCTL, a specification TA, or a regular expression. Can be used by model generator to optimise model for verification of specific properties.

Fig. 6.3: *CANDLE* Development Environment: Architecture

- ensures that all tools have a *consistent* view of a development project, so that the principle of 'What You Verify Is What You Execute' is respected.

The *CANDLE* development environment is intended to meet these requirements in supporting the development of CAN-based embedded systems. It allows the integration of a variety of tools for implementation and validation. A key aspect of the environment in maintaining consistency and promoting usability is its use of the same set of inputs for system implementation and model generation, as shown in Figure 6.3. It is intended that system implementation in *CANDLE* will follow a similar path to ESTEREL [Ber98b] and AORTA [Bra95]. This will be the subject of a future research project and is not considered further here. The remainder of this section is devoted to the model generation and analysis



`.code`	Source and object code files of implemented system; produced by system generation component and required by model generation component for code timing analysis.
`.c`	Source code of the LTS module; combined with *CANDLE* and OPEN/CÆSAR libraries to produce executable analysis program.
`.out`	Output of symbolic analysis: Reachability graph, Yes/No answer, Timed/Untimed diagnostic trace etc.

Fig. 6.4: *CANDLE* Validation Environment: Architecture

components of the validation environment.

6.5.2 Validation Environment

The validation environment of *CANDLE* consists of components for model generation and analysis. The core of the validation environment is organised according to the principles of the OPEN/CÆSAR architecture [Gar98]. This approach means that the CADP tool box [FGK$^+$96] and OPEN/KRONOS [Tri98] are immediately applicable to *CANDLE* programs; it also provides a very flexible mechanism for extending the validation environment in the future. The architecture of the *CANDLE* validation environment is illustrated in Figure 6.4. Its main features are discussed in more detail below.

6.5.3 The OPEN/CÆSAR Architecture

The design of the OPEN/CÆSAR architecture evolved during the course of projects to extend the functionality of the CÆSAR compiler [Gar92], a translator from LOTOS programs to labelled transition systems. Functional extensions included tools for random execution, interactive simulation, behavioural equivalence checking, temporal logic model-checking and test case generation. The desire to allow these tools to be used with languages other than LOTOS led to a design which encapsulates all language-dependent aspects. The final design offers a flexible and practical basis for the development of an open, extensible validation environment.

Encapsulation of language dependencies is achieved in OPEN/CÆSAR by requiring that any source program is seen by the validation environment as simply a labelled transition system which implements a well-defined application programming interface (API). The LTS API provides access to a representation of states and labels, and primitive operations to compute the transition relation (i.e., the initial state and successors of a given state). Knowledge of a source program by validation tools is restricted to the LTS API, which is implemented by a C program generated by an OPEN/CÆSAR-*compliant compiler* for the source language.

6.5.4 Model Generation

CANDLE-OPEN is the OPEN/CÆSAR-compliant compiler for *CANDLE* (Figure 6.4). It translates a *CANDLE* program into a C program which implements the OPEN/CÆSAR LTS API, generating the *CANDLE* program's simulation graph on demand. *CANDLE*-OPEN defines interfaces which integrate a number of loosely coupled components. These components are described briefly below.

- **Code Timing Analysis**: This component provides a connection to a program for calculating execution time bounds on sequential code fragments. This can be used to obtain bounds for the data operations and expressions of the *CANDLE* program by analysing their implementations. Access to the source and object code files of the implementation is provided by the system generation module (Figure 6.3). The process map and processor models are given by the system architecture files. So far, we have experimented with the use of the CINDERELLA code timing analysis tool for 68000 micro-processors [LMW95]. Other code timing tools remain to be investigated.

  In the case that the implementation of some data operation is not available for analysis, as will be the case quite often in the early stages of a design, execution time bounds are obtained from the `bounds` clause of the *SDML* model of the operation. The user can configure the development environment to obtain the bounds on each data operation either by analysis of its implementation, by examination of its `bounds` clause, or by user input via the keyboard. This allows the design to be explored in whatever way is judged to be most convenient or interesting.

- **Network Analysis**: This component constructs the static network model, which defines for each communication channel, its message set, message priorities and message transmission latencies. The message set and message priorities of each channel are stated explicitly in the *CANDLE* program. Calculation of transmission latencies depends upon the characteristics of the physical communication links implementing the channels and upon the size of message packets.

- **Data Abstraction**: This component constructs an abstract data model from a data specification. The abstract data model is described in *SDML* and currently is generated by the user. The integration into *CANDLE*-OPEN of a tool such as InVeSt [BLO98] to support the construction of the abstract model is envisaged.

- **Module Expansion**: This component constructs a 'flattened' *CANDLE* program by in-line expansion of all module instantiations.

- **CAN2BCAN, BCAN2C, SDML2C**: These components combine to generate the C program which implements the LTS module. CAN2BCAN translates the flattened *CANDLE* program to its equivalent *bCANDLE* representation, using the techniques described in §6.4. BCAN2C generates the C functions to implement the algorithm described in §5.2 to compute the transition relation of the simulation graph. SDML2C generates C functions to implement the data operations and predicates. As an alternative to generating code to construct the simulation graph on-the-fly, the user can choose to construct a TA in KRONOS `.tg` format, for later analysis. Although not yet implemented, other outputs which could be easily generated include UPPAAL `.ta` format [LPY97] and IF code [BFG$^+$99b]. These may give access to a wider range of analyses and optimisations.

Optimisation

A variety of optimisations can be applied at several stages of the model generation process, in order to combat state explosion. For example, *variable analysis* [BFG99a, SS98] can improve the quality of C functions generated by BCAN2C and SDML2C, by identifying dead variables which can be consistently reset. In addition, it seems possible to take advantage of *program slicing* techniques [CDH$^+$00, HDZ00, LS98], which can reduce the size of a model by removing those parts of it which cannot affect the outcome of the analysis of some specified property. Application of such optimisations remains to be investigated.

6.5.5 Model Exploration

A variety of tools can be applied to the exploration of generated LTS models. Many are provided by exploration modules from CADP or OPEN/KRONOS; the `generator` module has been developed specifically for *CANDLE*, in order to experiment with the MA state storage technique described in Chapter 5.

```
module Boiler is
   behaviour
     WaterLevel[Msecs(5)/WL_READY_PERIOD,
               Msecs(10)/WL_NORMAL_PERIOD, w1/w]
   |
     Pump[Msecs(5)/PUMP_READY_PERIOD]

   |
     Controller[Msecs(15)/SENSOR_TIMEDOUT, w2/w]
end module
```

Fig. 6.5: The Steam Boiler module

- `xsimulator` is an interactive simulator allowing step-by-step exploration of the simulation graph in a window-based environment.
- `evaluator` implements local and global algorithms for on-the-fly model-checking of branching μ-calculus.
- `exhibitor` performs a depth-first or breadth-first search for a finite untimed trail matching an input regular expression.
- `profounder` is an OPEN/KRONOS module which implements an algorithm to test for language emptiness of the simulation graph and a specification TBA.
- `generator` builds the simulation graph of the system, using one of several user-specified state storage mechanisms.

6.6 An example

In this section, we give an example of the use of *CANDLE* in modelling a slightly larger control system. The example is a modified version of the steam boiler control problem [ABL96].

6.6.1 The *CANDLE* program

We consider a system comprising a steam boiler, a pump, and a water-level sensor. We assume that the pump controls the flow of water into the boiler and that steam is drawn off via a steam outlet pipe. The water-level sensor gives the level of water in the tank. The purpose of the control program is to ensure that the water level is maintained within minimum and maximum bounds, or to shutdown the system if failure of the water-level sensor is detected.

The main program module is *Boiler*, shown in Figure 6.5. It shows that the system is structured as the parallel composition of three processes: *WaterLevel*, *Pump* and *Controller*, which are described in Figures 6.6 and 6.7. Figure 6.8 shows the data module for the system.

Each process executes three phases: *local initialisation*, *ready* and *normal*.

- In the local initialisation phase, a process resets its devices and initialises its local data. It is assumed that the system is started with the water level in the boiler between *low* and *high*, and the pump *off*.

```
module WaterLevel is
  const
    WL_READY_PERIOD  : duration;
    WL_NORMAL_PERIOD : duration
  type
    water_level
  procedure
    InitSensor(out water_level);
    ReadSensor(out water_level)
  channel
    k : (shutdown.unit,
         level.water_level,
         start.unit,
         sensor_ready.unit)
  var
    w : water_level
  behaviour
    InitSensor(w);
    select
    :: rcv(k,start)
    in
      every WL_READY_PERIOD do
        snd(k,sensor_ready)
      end every
    end select;
    select
    :: rcv(k,shutdown) ; idle
    in
      every WL_NORMAL_PERIOD do
        ReadSensor(w);
        snd(k,level.w)
      end every
    end select
end module
```

(a)

```
module Pump is
  const
    PUMP_READY_PERIOD :  duration
  type
    pump_status
  procedure
    InitPump(out pump_status);
    PumpOn(out pump_status);
    PumpOff(out pump_status)
  channel
    k : (shutdown.unit,
         pump_off.unit,
         pump_on.unit,
         start.unit,
         pump_ready.unit)
  var
    p : pump_status
  behaviour
    InitPump(p);
    select
    :: rcv(k,start)
    in
      every PUMP_READY_PERIOD do
        snd(k,pump_ready)
      end every
    end select;
    select
    :: rcv(k,shutdown); PumpOff(p); idle
    in
      loop do
        select
        :: rcv(k,pump_on) ; PumpOn(p)
        :: rcv(k,pump_off) ; PumpOff(p)
        end select
      end loop
    end select
end module
```

(b)

Fig. 6.6: Water-level Sensor and Pump modules

- In the ready phase, the *WaterLevel* and *Pump* processes repeatedly transmit a *ready* message until they receive a *start* message from the *Controller*. The *start* message is broadcast by *Controller* after it has received a *ready* message from both *Waterlevel* and *Pump*.

- In normal operation, *WaterLevel*, repeatedly reads the water-level sensor, updating a data variable with the current sensor value before broadcasting the value on channel k. The *Controller* process receives the sensor value from channel k and stores it in a data variable. *IsLowLevel* and *IsHighLevel* are boolean functions on the data state, used to test the value of the water level variable. If the level is too high, a message is sent to turn off the pump, if the level is too low, a message is sent to turn on the pump, otherwise the pump is left in its current state. If *Controller* does not receive a water-level message before timing out, it is assumed that the water-level sensor is faulty and a *shutdown* message is broadcast which brings the operation of the system to a halt with the pump turned off. We assume that the system is then made safe manually.

```
module Controller is
  const
    SENSOR_TIMEDOUT : duration
  type
    water_level
  procedure
    InitController(out water_level)
  function
    IsHighLevel(water_level) : boolean;
    IsLowLevel(water_level)  : boolean
  channel
    k : (shutdown.unit, pumpoff.unit, pumpon.unit,
         level.water_level, start.unit, pump_ready.unit,
         sensor_ready.unit)
  var
    w : water_level
  behaviour
    InitController(w);
    select
    :: rcv(k,sensor_ready); rcv(k,pump_ready)
    :: rcv(k,pump_ready); rcv(k,sensor_ready)
    end select;
    snd(k,start);
    loop do
      select
      :: rcv(k,level.w);
         if    IsHighLevel(w) then snd(k,pump_off)
         elsif IsLowLevel(w)  then snd (k,pump_on)
         end if
      :: elapse SENSOR_TIMEDOUT; snd(k,shutdown); idle
      end select
    end loop
end module
```

Fig. 6.7: Controller module

6.6.2 The *bCANDLE* model

The *CANDLE* program for the steam boiler can be translated into a *bCANDLE* model, as shown in Figure 6.9.

We have made a number of simplifying assumptions in order to clarify the relationship between the program and its model:

- Each process is allocated to its own dedicated processor.
- All processors run at the same speed, where 1 clock cycle is assumed to be 1 μsec, which is assumed to be equivalent to 1 unit of `duration`.
- The bus implementing channel k runs at $10^6\,bit/s$, i.e. 1 bit is transmitted in 1 μsec.
- The bounds on all *pre_snd*, *post_snd*, *pre_rcv* and *post_rcv* operations are 0, and so these operations have been omitted from the model.

The description of process behaviour in Figure 6.9 has been derived using the translation method described earlier, with the assumptions stated above and with some small modifications to aid readability: recursion is expressed using the equational style, rather than by using explicit rec terms; 'extra' equational

```
data Boiler is
type water_level is {low, ok, high}
type pump_status is {on, off}

procedure
  InitSensor(out wl : water_level) is bounds Cycles(300); Cycles(350)
    begin wl := ok end ;

  InitPump(out p : pump_status) is bounds Cycles(250); Cycles(1500)
    begin p := off end ;

  InitController(out wl : water_level) is bounds Cycles(400); Cycles(500)
    begin wl := ok end;

  PumpOn(out p : pump_status) is bounds Cycles(200); Cycles(300)
    begin p := on end;

  PumpOff(out p : pump_status) is bounds Cycles(200); Cycles(300)
    begin p := off end

  ReadSensor(out wl : water_level) is bounds Cycles(50); Cycles(75)
    begin wl := any water_level end

function
  IsHighLevel(wl : water_level) : boolean is bounds Cycles(10); Cycles(15)
    begin return (wl = high) end;

  IsLowLevel(wl : water_level) : boolean is bounds Cycles(10); Cycles(15)
    begin return (wl = low) end
end data
```

Fig. 6.8: Steam Boiler Data Module

definitions have been introduced to emphasise the initialisation, ready and normal phases of process behaviour.

The network section of the model is derived from the *CANDLE* `channel` declarations and the assumptions about the underlying communication mechanism. It defines the network structure – in this case, simply a single channel – giving the priority of messages and the transmission latency function. Notice that all messages, except *level* messages, consist only of a message identifier, whereas *level* messages contain a water level value in addition to the message identifier, and hence have a greater pre-acceptance latency.

The data section declares the names of the data variables used in the model. The *bCANDLE* data environment is constructed from the *SDML* data module in a straightforward way, as described in §6.4.3. We do not elaborate the data environment here.

6.6.3 Analysis of the model

The *CANDLE*-OPEN environment can be used to generate the simulation graph of the *bCANDLE* model, and to explore it interactively, or exhaustively, to ensure that it exhibits desirable behaviour. In this case, `generator` produces the graph in less than a second on a 233MHz Pentium II, running RedHat Linux 5.2. The output

```
WaterLevel | Pump | Controller

where

WaterLevel = [InitSensor: 300,350]; WL_Ready; WL_Normal
WL_Ready = WL_Ready0 [> k?start._
WL_Ready0 = k!sensor_ready._ ; idle
            [> [WL_ReadyPeriod: 5000,5100] ; WL_Ready0
WL_Normal = (WL_Normal0 [> k?shutdown._ ; idle)
WL_Normal0 =
    [ReadWaterLevel:50,75]; k!level.w1 ; idle
    [>
    [WL_NormalPeriod:10000,10250]; WL_Normal0

Pump = [InitPump:250,1500]; P_Ready; P_Normal
P_Ready = P_Ready0 [> k?start._
P_Ready0 = k!pump_ready._ ; idle
            [> [P_ReadyPeriod: 5000,5100] ; P_Ready0
P_Normal = (P_Normal0 [> k?shutdown._; [PumpOff:200,300] ; idle)
P_Normal0 = (k?pumpon._ ; [PumpOn:200,300] +
             k?pumpoff._ ; [PumpOff:200,300]);
            P_Normal0

Controller = [InitController:400,500]; C_Ready; C_Normal
C_Ready = (k?sensor_ready._ ; k?pump_ready._ +
           k?pump_ready._ ; k?sensor_ready._);
           k!start._
C_Normal = k?level.w2;
             [TestHighLevel: 10,15];
             (HighLevel -> k!pumpoff._ +
              notHighLevel ->
                 [TestLowLevel: 10,15];
                 (LowLevel -> k!pumpon._ +
                  notLowLevel -> null));
             C_Normal
           +
           [SensorTimedOut: 15000,15500]; k!shutdown._ ; idle

network
/*                  Pri   dlb  dub  dlB duB   */
k = (shutdown       : 1,   35,  43,  11, 13;
     pumpoff        : 2,   35,  43,  11, 13;
     pumpon         : 3,   35,  43,  11, 13;
     level          : 4,   43,  53,  11, 13;
     start          : 5,   35,  43,  11, 13;
     pump_ready     : 6,   35,  43,  11, 13;
     sensor_ready   : 7,   35,  43,  11, 13
     )
data w1, w2, p
```

Fig. 6.9: A *bCANDLE* model for a simple boiler controller

```
generator: Release 1.1.1 Thu Oct 27 22:03:15 GMT 2000
Simulation Graph: Boiler
#states                 1963
#trans                  2661
#matrices               1123
```

shows the number of states and transitions in the simulation graph. The number of matrices indicates the number of distinct clock zones explored. It is worth noting that the simulation graph for the boiler control system is very much smaller than the equivalent TA generated from the *bCANDLE* model using the standard techniques of Chapter 4. That TA has more than 500,000 locations, exceeding the capacity of model checkers such as KRONOS. This is a clear indication of the benefit of generating the simulation graph 'on-the-fly'. The application of clock activity reduction to the example reduces the number of clocks required from 12 to 5. This also has a significant effect on the size of the model, reducing the number of states in the simulation graph from $> 115,000$ to 1963.

The simulation graph can be checked for a variety of properties, which increase confidence in the correctness of the control system. A number of simple properties, which the graph satisfies, are discussed below.

Notation. Notice that the properties are expressed using predicates over the state variables, rather than using a propositional encoding as required by KRONOS. The status and pending message queue of a channel k are denoted $k.status$ and $k.queue$, respectively. See §3.4.1 for an explanation of other channel notation.

1. Basic 'sanity' checks:

   (a) The model is non-Zeno.
   $init \Rightarrow \forall\Box\, \exists\Diamond_{=1}\, \mathbf{true}$

   (b) Whenever the channel is not busy, and there are messages pending transmission, the channel begins transmitting a message immediately.
   $\forall\Box((k.status = \mathsf{FREE} \wedge k.queue \neq \langle\rangle) \Rightarrow \forall\Diamond_{=0}\, k.status = \mathsf{PRE})$

   In fact, it can be shown that any persistent *bCANDLE* model satisfies these properties. However, failure to satisfy either property alerts the user to a fundamental error in the construction of the model.

2. Properties of the communication system:

   (a) A pending message is never overwritten, i.e. once a message is queued, it will be transmitted before another message with the same identifier is queued.
   $\forall\Box(\mathsf{enable}(k!i.\_) \Rightarrow \forall j \in \mathrm{dom}(k.queue)\ .\ k.queue[j] \neq i.\_)$

   (b) When any message transmission reaches the acceptance point, some process will be able to accept the message.
   $\forall\Box(\mathsf{enable}(k\!\uparrow m) \Rightarrow \forall\Diamond_{=0}\, \mathsf{enable}(k?m))$

(c) The time between acceptance tests for any type of message is at least t.

$$\forall\Box(\mathsf{enable}(k{\uparrow}\,i.\_) \Rightarrow \forall\Diamond_{=0}(\forall\Box_{<t}\neg\ \mathsf{enable}(k{\uparrow}\,i.\_)))$$

Of course, there are many systems for which these properties are not required. But, very often, the failure of one or more of them is an indication of a flaw in the implementation of the control system. For example, property (c) is helpful in checking the behaviour of a multi-tasking node implemented with round robin scheduling and polled communication, as the value of t for any channel and message type should not be less than the quantum of the scheduler, otherwise messages may be lost.

3. Basic response properties:

(a) If a *low* water level is detected, then the pump will be *on* within 1msec.

$$\forall\Box(w1 = low \Rightarrow \forall\Diamond_{\leq 1000}(p = on))$$

(b) If a *high* water level is detected, then the pump will be *off* within 1msec.

$$\forall\Box(w1 = high \Rightarrow \forall\Diamond_{\leq 1000}(p = off))$$

4. Further response properties:

(a) If it is not possible for the *Controller* to receive a *level* message immediately, then, within 16msecs, it can receive such a message, or it will transmit the *shutdown* message.

$$\forall\Box(\neg\ \mathsf{enable}(k?level.\_) \Rightarrow$$
$$\forall\Diamond_{\leq 16000}(\mathsf{enable}(k?level.\_) \vee \mathsf{enable}(k!shutdown)))$$

(b) If transmission of the *shutdown* message is enabled, then, within 1msec, the pump is turned *off* and the system idles.

$$\forall\Box(\mathsf{enable}(k!shutdown) \Rightarrow \forall\Diamond_{\leq 1000}(p = off \wedge \forall\Box(\neg\ \mathsf{enable}(\_))))$$

Verification of some control system properties is most conveniently undertaken by an analysis of the control system model in conjunction with a model of its environment. For example, to verify that the boiler control system always maintains the level of water within acceptable bounds, a model of the boiler can be constructed, including aspects of its behaviour such as: the rate of flow of water from the pump; the response lag of the pump to a control command; the rate of flow of steam from the boiler, and so on. Henzinger and Wong-Toi [HWT96] describe a hybrid automata model of a steam boiler, which could form the basis of a suitable environmental model for composition with our model of the boiler control system. As usual, the limiting factor is the state explosion problem. A benefit of our approach is that it is possible to take advantage of a minimisation tool, such as minim [Tri98], in order to reduce the size of the control system model before composing it with an environmental model.

6.7 Conclusions and Related Work

6.7.1 Conclusions

We have defined a programming language for broadcasting embedded control systems. The language has many of the constructs which one would expect in a modern, real-time language [BW01], and it has been shown that the process language *bCANDLE* is expressive enough to give a semantics to these constructs in a natural way. We have described a development and validation environment which integrates a variety of languages and tools. In particular, the environment supports the translation of *CANDLE* programs to TA, enabling the application of tools such as KRONOS and CADP for validation. Future work will include further development of the language, in particular to improve the module system, and also further development of the tool support. Automation of the development of executable abstract data models from B or VDM specifications is also of interest. An important focus of future work will be the efficient composition of control system models with environmental models in order to allow validation of a wider range of system properties. Finally, more experience is needed of applying both language and tools to a wider range of case studies, so that the techniques can be tested on examples which are more realistic in terms of their size and complexity than those which are described in this chapter.

6.7.2 Related Work

ESTEREL [BG92] is the classical example of a language which supports both the development and validation of embedded systems. It represents the synchronous approach which has been so effective in the uniprocessor domain. Interest in the application of model-checking to asynchronous systems is comparatively recent. One of the first tools for untimed systems is VeriSoft [God97] which supports stateless search in the verification of C programs. Holzmann and Smith [HS99] describe an approach in which a SPIN [Hol97] model is extracted from an annotated C program, allowing the checking of properties specified in LTL. Related approaches to the verification of Java programs are described by Havelund and Pressburger [HP00] and Corbett et al. [CDH$^+$00]. The latter work builds upon experience gained in applying similar techniques to the verification of Ada programs [DPC98]. Huch [Huc99] has developed a dedicated model checker for a subset of Erlang – an untyped higher-order concurrent functional language with asynchronous communication primitives. All of these approaches employ techniques to construct an abstract model using only the source code of the program to be verified. Our approach enforces a clear separation of control algorithms and sequential data operations, and assumes that abstract specifications are available for the latter. We believe that these specifications will prove to be a better starting point for the construction of efficient models.

In the case of timed systems, the work of Corbett [Cor96] is the most ambitious in its choice of input language. He describes a method for translating (a subset of) Ada programs into hybrid automata, so enabling the checking of a variety of temporal properties using the HyTech [HHWT97] model checker. His models can accommodate fixed priority pre-emptive scheduling of concurrent tasks but he considers only uniprocessor systems, deferring distributed systems to further work. Hune [Hun99] considers the problem of using UPPAAL [LPY97] for verifying programs executing on the LEGO RCX brick. The LEGO RCX brick is part of the LEGO MINDSTORMS range of LEGO toys. It contains a micro-processor and is equipped with three sensors, three actuators and an infra-red port for communication to enable program downloading. Hune describes a translation from the RCX assembly language into a network of TA described using the UPPAAL input language. Again, the work is restricted to uniprocessor systems; the scheduling policy is round-robin. Iversen et al. [IKL$^+$00] also consider the same problem, but allow the use of a more expressive programming language, NQC (Not Quite C), which is a restricted form of C. Dierks [Die01] introduces PLC-automata with a view to developing and analysing real-time systems implemented with programmable logic controllers. He describes a method for translating PLC-automata to timed automata so that KRONOS and UPPAAL can be used for the analysis.

This brief survey of related work is evidence of the recent interest in applying model-checking to the analysis of embedded system implementations. We believe that the work described in this chapter is the first to present a general method for real-time model-checking of distributed, high-level programs implemented on an industry-standard, broadcast network.

7. CONCLUSIONS AND FURTHER WORK

7.1 Conclusions

Formal methods can be useful for gaining confidence in the correct behaviour of systems. Expressive languages and automatic analysis techniques are needed to promote the acceptance of formal methods in industry. For embedded systems, such languages and techniques should allow the expression of, and reasoning about, real-time properties. For a large class of embedded systems, broadcast communication is an implementation primitive and should be accommodated comfortably within a formal method intended for application in that domain. This dissertation has proposed a formal language which is claimed to satisfy these requirements, at least partially. Our approach has been to define a language in which process behaviour can be described using a few primitive operators, including operators for the sending and receiving of broadcast messages. The communication semantics is an abstraction of the CAN protocol and models both message priority and communication latency. This language has proved suitable as a bridge between the high-level expression of embedded system models and their low-level representation in a form which is appropriate for the application of a wide variety of analysis techniques. We have demonstrated how the most efficient of current methods can be applied to models expressed in our language. In particular, we have given an algorithm for on-the-fly generation of the simulation graph, including clock activity reduction. This provides a foundation for the application of methods such as reachability analysis, model checking, TBA emptiness, minimisation and time-abstract bisimulation, as implemented in tools such as KRONOS, UPPAAL and CADP. In addition, we have demonstrated the use of minimised automata for compact state space representation. Minimised automata have been employed in the model checking tool SPIN for the analysis of untimed, asynchronous models. They are applied here, for the first time, in the analysis of timed system models and we give experimental data to confirm their utility.

7.2 Further Work

The expressiveness of our language is restricted in several ways. For example, we do not allow control to depend explicitly on the time of event occurrences, nor is it possible for an interrupted task to resume execution from its point of interruption. Both features are available in ET-LOTOS [Her98], for example.

More seriously, we cannot model multi-tasking systems in which the CPU resource is allocated to tasks using a more sophisticated scheduling policy than round-robin, e.g., a fixed priority preemptive policy. It also remains to consider the modelling of the occurrence of faults in broadcast message transmission.

It is not difficult to see how some of these additional features could be accommodated, e.g. an explicit scheduler could be added to the execution model, as could a 'daemon' for fault injection. The problem is to cope with the extra complexity and its effect on state explosion. The addition of such features almost certainly leads to a hybrid system model for which many verification problems become either undecidable or, at best, even more resource demanding [HKPV95].

Even without adding to the complexity of the language, further work is needed on state space explosion. Some obvious lines of inquiry are suggested at the end of Chapter 5, where work on variable ordering and live variable analysis has the potential to bring reductions in the size of the discrete state space. Also of interest is investigation of the use of partial order reduction and symbolic clock constraint representations. In particular, research is needed to compare the performance of DDD's with that of MA's when applied to typical asynchronous broadcast systems, especially when considered in combination with reduction techniques such as partial order and inclusion/convex hull abstractions, where the use of MA's seems to offer a *prima facie* advantage.

One can imagine that more use can be made of the high-level, algebraic structure apparent in the models, to transform them into more space-efficient, equivalent representations. This should be possible at all levels, from the high-level *CANDLE* model, through the *bCANDLE* and net representations, to the timed automaton. Undoubtedly, compositional techniques will be required in order to extend a fully automated approach to industrial-scale systems.

Finally, further work remains on a number of pragmatic issues affecting industrial usage of the technology: at a high-level, the issue of requirements capture and their relationship to formal specifications; at a low-level, the formal specification and implementation of an execution environment which satisfies the abstraction assumptions of Chapter 3.

Some progress has been made but much remains to be done before it will be possible to realise Pnueli's vision of a *seamless development process* [Pnu99] for broadcasting embedded control systems.

APPENDIX

A. FLOW REGULATOR TA

A.1 KRONOS `.tg` Format

The KRONOS `.tg` format adopts the following conventions in the presentation of a TA. Each location is identified by a unique integer introduced by the keyword `state`. The location invariant is shown following the keyword `invar` and outgoing edges following the keyword `trans`. Each edge is of the form `guard` ⇒ `label ; reset H; goto target`, where ⇒, `reset` and `goto` are keywords, `guard` is a clock condition, `label` is the edge label, `H` is a set of clocks and `target` is an integer identifying the target location of the edge. The *bCANDLE* translator introduces further conventions with respect to the structure of labels: communications of the form $k!i.x$ and $k?i.x$ are labelled `SND_k_i_v` and `RCV_k_i_v`, respectively, where `k` and `i` are shown as their internal integer representations and `v` is the value of `x` in the current data environment. Similarly, the network action labels, $k \rightsquigarrow i.v$, $k\uparrow i.v$, $i.v \rightsquigarrow k$, and $k\downarrow$ are shown as `FP_k_i_v`, `PA_k_i_v`, `AP_k`, and `PF_k`, respectively. Operation names are prefixed with `OP_` and predicate names with `PRED_`.

A.2 Flow Regulator TA

```
#states                  48
#trans                  146
#clocks                   5
H1 H2 H3 H4 H5



state: 0
invar: H3 <= 90 AND H5 <= 10250
trans:
H3 >= 85 => OP_ReadSensor; RESET{ H1 }; goto 1
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 0

state: 1
invar: H1 <= 0 AND H5 <= 10250
trans:
true => SND_0_0_0; RESET{ H1 }; goto 2
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 0

state: 2
invar: H1 <= 0 AND H5 <= 10250
trans:
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 3
true => FP_0_0_0; RESET{ H1 H2 }; goto 4

state: 3
invar: H1 <= 0 AND H3 <= 90 AND H5 <= 10250
trans:
H3 >= 85 => OP_ReadSensor; RESET{ H1 }; goto 5
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 3
true => FP_0_0_0; RESET{ H1 H2 }; goto 6

state: 4
invar: H2 <= 53 AND H5 <= 10250
trans:
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 6
H2 >= 43 => PA_0_0_0; RESET{ H1 }; goto 7

state: 5
invar: H1 <= 0 AND H1 <= 0 AND H5 <= 10250
trans:
true => SND_0_0_0; RESET{ H1 }; goto 2
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 3
true => FP_0_0_0; RESET{ H1 H2 }; goto 8

state: 6
invar: H2 <= 53 AND H3 <= 90 AND H5 <= 10250
trans:
H3 >= 85 => OP_ReadSensor; RESET{ H1 }; goto 8
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 6
H2 >= 43 => PA_0_0_0; RESET{ H1 }; goto 9

state: 7
invar: H1 <= 0 AND H5 <= 10250
trans:
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 9
true => RCV_0_0_0; RESET{ H1 H4 }; goto 10

state: 8
invar: H2 <= 53 AND H1 <= 0 AND H5 <= 10250
trans:
true => SND_0_0_0; RESET{ H1 }; goto 11
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 6
H2 >= 43 => PA_0_0_0; RESET{ H1 }; goto 12

state: 9
invar: H1 <= 0 AND H3 <= 90 AND H5 <= 10250
trans:
H3 >= 85 => OP_ReadSensor; RESET{ H1 }; goto 12
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 9
true => RCV_0_0_0; RESET{ H1 H4 }; goto 13

state: 10
invar: H1 <= 0 AND H5 <= 10250 AND H4 <= 300
trans:
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 13
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 7
true => AP_0; RESET{ H1 H2 }; goto 14

state: 11
invar: H2 <= 53 AND H5 <= 10250
trans:
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 15
H2 >= 43 => PA_0_0_0; RESET{ H1 }; goto 16

state: 12
invar: H1 <= 0 AND H1 <= 0 AND H5 <= 10250
trans:
true => SND_0_0_0; RESET{ H1 }; goto 16
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 9
true => RCV_0_0_0; RESET{ H1 H4 }; goto 17

state: 13
invar: H1 <= 0 AND H3 <= 90 AND H5 <= 10250 AND H4 <= 300
trans:
H3 >= 85 => OP_ReadSensor; RESET{ H1 }; goto 17
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 13
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 9
true => AP_0; RESET{ H1 H2 }; goto 18

state: 14
invar: H2 <= 12 AND H5 <= 10250 AND H4 <= 300
trans:
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 18
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 19
H2 >= 10 => PF_0; RESET{ H1 }; goto 20

state: 15
invar: H2 <= 53 AND H3 <= 90 AND H5 <= 10250
trans:
H3 >= 85 => OP_ReadSensor; RESET{ H1 }; goto 21
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 15
H2 >= 43 => PA_0_0_0; RESET{ H1 }; goto 22

state: 16
invar: H1 <= 0 AND H5 <= 10250
trans:
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 22
true => RCV_0_0_0; RESET{ H1 H4 }; goto 23

state: 17
invar: H1 <= 0 AND H1 <= 0 AND H5 <= 10250 AND H4 <= 300
trans:
true => SND_0_0_0; RESET{ H1 }; goto 23
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 13
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 12
```

```
true => AP_0; RESET{ H1 H2 }; goto 24

state: 18
invar: H2 <= 12 AND H3 <= 90 AND H5 <= 10250 AND H4 <= 300
trans:
H3 >= 85 => OP_ReadSensor; RESET{ H1 }; goto 24
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 18
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 25
H2 >= 10 => PF_0; RESET{ H1 }; goto 26

state: 19
invar: H2 <= 12 AND H5 <= 10250
trans:
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 25
H2 >= 10 => PF_0; RESET{ H1 }; goto 27

state: 20
invar: H5 <= 10250 AND H4 <= 300
trans:
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 26
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 27

state: 21
invar: H2 <= 53 AND H1 <= 0 AND H5 <= 10250
trans:
true => SND_0_0_0; RESET{ H1 }; goto 11
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 15
H2 >= 43 => PA_0_0_0; RESET{ H1 }; goto 28

state: 22
invar: H1 <= 0 AND H3 <= 90 AND H5 <= 10250
trans:
H3 >= 85 => OP_ReadSensor; RESET{ H1 }; goto 28
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 22
true => RCV_0_0_0; RESET{ H1 H4 }; goto 29

state: 23
invar: H1 <= 0 AND H5 <= 10250 AND H4 <= 300
trans:
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 29
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 16
true => AP_0; RESET{ H1 H2 }; goto 30

state: 24
invar: H2 <= 12 AND H1 <= 0 AND H5 <= 10250 AND H4 <= 300
trans:
true => SND_0_0_0; RESET{ H1 }; goto 30
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 18
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 31
H2 >= 10 => PF_0; RESET{ H1 }; goto 32

state: 25
invar: H2 <= 12 AND H3 <= 90 AND H5 <= 10250
trans:
H3 >= 85 => OP_ReadSensor; RESET{ H1 }; goto 31
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 25
H2 >= 10 => PF_0; RESET{ H1 }; goto 0

state: 26
invar: H3 <= 90 AND H5 <= 10250 AND H4 <= 300
trans:
H3 >= 85 => OP_ReadSensor; RESET{ H1 }; goto 32
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 26
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 0

state: 27
invar: H5 <= 10250
trans:
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 0

state: 28
invar: H1 <= 0 AND H1 <= 0 AND H5 <= 10250
trans:
true => SND_0_0_0; RESET{ H1 }; goto 16
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 22
true => RCV_0_0_0; RESET{ H1 H4 }; goto 33

state: 29
invar: H1 <= 0 AND H3 <= 90 AND H5 <= 10250 AND H4 <= 300
trans:
H3 >= 85 => OP_ReadSensor; RESET{ H1 }; goto 33
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 29
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 22
true => AP_0; RESET{ H1 H2 }; goto 34

state: 30
invar: H2 <= 12 AND H5 <= 10250 AND H4 <= 300
trans:
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 34
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 35
H2 >= 10 => PF_0; RESET{ H1 }; goto 36

state: 31
invar: H2 <= 12 AND H1 <= 0 AND H5 <= 10250
trans:
true => SND_0_0_0; RESET{ H1 }; goto 35
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 25
H2 >= 10 => PF_0; RESET{ H1 }; goto 1

state: 32
invar: H1 <= 0 AND H5 <= 10250 AND H4 <= 300
trans:
true => SND_0_0_0; RESET{ H1 }; goto 36
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 26
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 1

state: 33
invar: H1 <= 0 AND H1 <= 0 AND H5 <= 10250 AND H4 <= 300
trans:
true => SND_0_0_0; RESET{ H1 }; goto 23
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 29
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 28
true => AP_0; RESET{ H1 H2 }; goto 37

state: 34
invar: H2 <= 12 AND H3 <= 90 AND H5 <= 10250 AND H4 <= 300
trans:
H3 >= 85 => OP_ReadSensor; RESET{ H1 }; goto 37
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 34
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 38
H2 >= 10 => PF_0; RESET{ H1 }; goto 39

state: 35
invar: H2 <= 12 AND H5 <= 10250
trans:
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 38
H2 >= 10 => PF_0; RESET{ H1 }; goto 2

state: 36
invar: H1 <= 0 AND H5 <= 10250 AND H4 <= 300
trans:
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 39
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 2
true => FP_0_0_0; RESET{ H1 H2 }; goto 40

state: 37
invar: H2 <= 12 AND H1 <= 0 AND H5 <= 10250 AND H4 <= 300
trans:
true => SND_0_0_0; RESET{ H1 }; goto 30
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 34
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 41
H2 >= 10 => PF_0; RESET{ H1 }; goto 42

state: 38
invar: H2 <= 12 AND H3 <= 90 AND H5 <= 10250
trans:
H3 >= 85 => OP_ReadSensor; RESET{ H1 }; goto 41
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 38
H2 >= 10 => PF_0; RESET{ H1 }; goto 3

state: 39
invar: H1 <= 0 AND H3 <= 90 AND H5 <= 10250 AND H4 <= 300
trans:
H3 >= 85 => OP_ReadSensor; RESET{ H1 }; goto 42
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 39
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 3
true => FP_0_0_0; RESET{ H1 H2 }; goto 43

state: 40
invar: H2 <= 53 AND H5 <= 10250 AND H4 <= 300
trans:
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 43
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 4
H2 >= 43 => PA_0_0_0; RESET{ H1 }; goto 10

state: 41
invar: H2 <= 12 AND H1 <= 0 AND H5 <= 10250
trans:
true => SND_0_0_0; RESET{ H1 }; goto 35
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 38
H2 >= 10 => PF_0; RESET{ H1 }; goto 5

state: 42
invar: H1 <= 0 AND H1 <= 0 AND H5 <= 10250 AND H4 <= 300
trans:
true => SND_0_0_0; RESET{ H1 }; goto 36
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 39
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 5
true => FP_0_0_0; RESET{ H1 H2 }; goto 44
```

```
state: 43
invar: H2 <= 53 AND H3 <= 90 AND H5 <= 10250 AND H4 <= 300
trans:
H3 >= 85 => OP_ReadSensor; RESET{ H1 }; goto 44
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 43
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 6
H2 >= 43 => PA_0_0_0; RESET{ H1 }; goto 13

state: 44
invar: H2 <= 53 AND H1 <= 0 AND H5 <= 10250 AND H4 <= 300
trans:
true => SND_0_0_0; RESET{ H1 }; goto 45
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 43
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 8
H2 >= 43 => PA_0_0_0; RESET{ H1 }; goto 17

state: 45
invar: H2 <= 53 AND H5 <= 10250 AND H4 <= 300
trans:
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 46
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 11
H2 >= 43 => PA_0_0_0; RESET{ H1 }; goto 23

state: 46
invar: H2 <= 53 AND H3 <= 90 AND H5 <= 10250 AND H4 <= 300
trans:
H3 >= 85 => OP_ReadSensor; RESET{ H1 }; goto 47
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 46
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 15
H2 >= 43 => PA_0_0_0; RESET{ H1 }; goto 29

state: 47
invar: H2 <= 53 AND H1 <= 0 AND H5 <= 10250 AND H4 <= 300
trans:
true => SND_0_0_0; RESET{ H1 }; goto 45
H5 >= 10000 => OP_PERIOD; RESET{ H1 H3 H5 }; goto 46
H4 >= 200 => OP_AdjustValve; RESET{ H1 }; goto 21
H2 >= 43 => PA_0_0_0; RESET{ H1 }; goto 33
```

B. PROOFS

B.1 Correctness of the translation

This section is concerned with demonstrating the correctness of the translation of *bCANDLE* system models to timed automata. Its purpose is to prove the validity of Proposition 4.2, which we state again here.

Proposition B.1 *Let $\widehat{B} \in b\widehat{CAN}$ be a clocked bCANDLE system and $B \mathrel{\widehat{=}} \mathsf{unclk}(\widehat{B})$ the corresponding unclocked system. Let $\mathcal{G}(\widehat{B})$ be the TA given by Definition 4.15. Then, the transition systems of $\mathcal{G}(\widehat{B})$ and B are strongly equivalent.*

$$\mathcal{T}\,[\![\,\mathcal{G}(\widehat{B})]\!] \mathrel{\underline{\leftrightarrow}} \mathcal{T}\,[\![B]\!]$$

The proof of the proposition depends upon demonstrating the existence of a strong bisimulation relation between $\mathcal{T}\,[\![\,\mathcal{G}(\widehat{B})]\!]$ and $\mathcal{T}\,[\![B]\!]$. A number of auxiliary definitions and lemmas are required.

Firstly, we define the set of states which can occur in the transition system of a clocked *bCANDLE* system, where it is required that in any such state $(\widehat{B}, \mathbf{v})$, the clock valuation $\mathbf{v}$ satisfies the location invariant $I(\widehat{B})$.

Definition B.1 Let $\mathcal{H}$ be a set of clock variables and $\widehat{B} \in b\widehat{CAN}$ a clocked *bCANDLE* system whose clocks are taken from the set $\mathcal{H}$, i.e. $\mathsf{clk}(\widehat{B}) \subseteq \mathcal{H}$. It is assumed that $\mathcal{H}$ also contains the distinguished urgent clock h_u, i.e. $h_u \in \mathcal{H} \setminus \mathsf{clk}(\widehat{B})$. We denote by $\Sigma_{b\widehat{CAN},\mathcal{H}}$ the states of the transition system of the TA for $\widehat{B}$, i.e.

$$\Sigma_{b\widehat{CAN},\mathcal{H}} \mathrel{\widehat{=}} \{(\widehat{B}, \mathbf{v}) \mid \widehat{B} \in b\widehat{CAN} \wedge \mathbf{v} \in \mathbb{R}^{\mathcal{H}} \wedge \mathbf{v} \models I(\widehat{B})\} \qquad \square$$

Now, a clocked *bCANDLE* system $\widehat{B}$ is related to an equivalent *bCANDLE* system B, by the notion of *aging*:

Definition B.2 (Aging) Let $\mathcal{H}$ be a set of clock variables. Let $\widehat{B} \in b\widehat{CAN}$ be a clocked *bCANDLE* system where $\mathsf{clk}(\widehat{B}) \subseteq \mathcal{H}$. Then, $\mathsf{age} : \Sigma_{b\widehat{CAN},\mathcal{H}} \to bCAN$ is a function giving an aged *bCANDLE* system, where

$$\mathsf{age}((\widehat{P}, \widehat{N}, D), \mathbf{v}) \mathrel{\widehat{=}} (\mathsf{age}(\widehat{P}, \mathbf{v}), \mathsf{age}(\widehat{N}, \mathbf{v}), D).$$

The function $\mathsf{age} : \widehat{Proc} \times \mathbb{R}^{\mathcal{H}} \to Proc$ is defined by

$$\begin{array}{rcl}
\mathsf{age}(k!i.x, \mathbf{v}) & \widehat{=} & k!i.x \\
\mathsf{age}(k?i.x, \mathbf{v}) & \widehat{=} & k?i.x \\
\mathsf{age}([\omega : t_1, t_2]^h, \mathbf{v}) & \widehat{=} & [\omega : t_1 \dot{-} \mathbf{v}(h), t_2 \dot{-} \mathbf{v}(h)] \\
\mathsf{age}(\gamma \to \widehat{P}, \mathbf{v}) & \widehat{=} & \gamma \to \mathsf{unclk}(\widehat{P}) \\
\mathsf{age}(\widehat{P} \,;\, \widehat{Q}, \mathbf{v}) & \widehat{=} & \mathsf{age}(\widehat{P}, \mathbf{v}) \,;\, \mathsf{unclk}(\widehat{Q}) \\
\mathsf{age}(\widehat{P} \bowtie \widehat{Q}, \mathbf{v}) & \widehat{=} & \mathsf{age}(\widehat{P}, \mathbf{v}) \bowtie \mathsf{age}(\widehat{Q}, \mathbf{v}), \qquad \bowtie \in \{+, [>, |\} \\
\mathsf{age}(\mathrm{rec}\, X.\widehat{P}, \mathbf{v}) & \widehat{=} & \mathsf{age}(\widehat{P}[\mathrm{rec}\, X.\widehat{P}/X], \mathbf{v})
\end{array}$$

As usual, we rely on the fact that $\widehat{P}$ is guarded to ensure that $\mathsf{age}(\widehat{P}, \mathbf{v})$ is well-defined.

The function $\mathsf{age} : \widehat{Network_K} \times \mathbb{R}^{\mathcal{H}} \to Network_K$ is defined by

$$\mathsf{age}(\widehat{N}, \mathbf{v}) \mathrel{\widehat{=}} \{k \mapsto \mathsf{age}(\widehat{N}_k, \mathbf{v}) \mid k \in K\}$$

where

$$\begin{array}{rcl}
\mathsf{age}((\downarrow, u)^h, \mathbf{v}) & \widehat{=} & (\downarrow, u) \\
\mathsf{age}((\overset{t_1,t_2}{\rightsquigarrow} m, u)^h, \mathbf{v}) & \widehat{=} & (\overset{t_1 \dot{-} \mathbf{v}(h), t_2 \dot{-} \mathbf{v}(h)}{\rightsquigarrow} m, u) \\
\mathsf{age}((\uparrow m, u)^h, \mathbf{v}) & \widehat{=} & (\uparrow m, u) \\
\mathsf{age}((m \overset{t_1,t_2}{\rightsquigarrow}, u)^h, \mathbf{v}) & \widehat{=} & (m \overset{t_1 \dot{-} \mathbf{v}(h), t_2 \dot{-} \mathbf{v}(h)}{\rightsquigarrow}, u)
\end{array}$$

□

The main proof makes use of the standard technique of demonstrating a strong bisimulation *up to* $\underline{\leftrightarrow}$. It is necessary to adapt the usual notion of strong bisimulation up to $\underline{\leftrightarrow}$ to $\approx$-bisimulation.

Definition B.3 (Strong bisimulation up to $\underline{\leftrightarrow}$) Let $\mathcal{S} = (\Sigma, \sigma^{\mathcal{I}}, L, \longrightarrow)$ be a LTS. Let $\approx$ be an equivalence relation on Σ. A binary relation $R \subseteq \Sigma \times \Sigma$ is a strong $\approx$-*bisimulation* up to $\underline{\leftrightarrow}$ if $\sigma_1 R \sigma_2$ implies

1. $\sigma_1 \approx \sigma_2$
2. for all $\lambda \in L$, if $\sigma_1 \xrightarrow{\lambda} \sigma_1'$, then $\sigma_2 \xrightarrow{\lambda} \sigma_2'$ for some σ_2' such that $\sigma_1' \,\underline{\leftrightarrow}\, R \,\underline{\leftrightarrow}\, \sigma_2'$
3. for all $\lambda \in L$, if $\sigma_2 \xrightarrow{\lambda} \sigma_2'$, then $\sigma_1 \xrightarrow{\lambda} \sigma_1'$ for some σ_1' such that $\sigma_1' \,\underline{\leftrightarrow}\, R \,\underline{\leftrightarrow}\, \sigma_2'$ □

Proposition B.2 *If R is a strong $\approx$-bisimulation up to $\underline{\leftrightarrow}$, then $\underline{\leftrightarrow}\, R \,\underline{\leftrightarrow}$ is a strong $\approx$-bisimulation.*

Proof As the proof of Lemma 4.5 in Milner [Mil89]. □

In order to apply the notion of strong bisimulation in the context of the main proof, it is necessary to extend $\approx_{ND}$-bisimulation to clocked states and to pairs of clocked and unclocked states. This requires us to revise our definition of context equivalence.

Definition B.4 (Context Equivalence) Let $\sigma_1, \sigma_2 \in bCAN \cup \Sigma_{\widehat{bCAN},\mathcal{H}}$ be either clocked or unclocked *bCANDLE* system states. We denote by $\sigma_1 \approx_{ND} \sigma_2$ that σ_1 is *context equivalent* to σ_2, and define $\approx_{ND} \subseteq bCAN \cup \Sigma_{\widehat{bCAN},\mathcal{H}} \times bCAN \cup \Sigma_{\widehat{bCAN},\mathcal{H}}$ by requiring that $\sigma_1 \approx_{ND} \sigma_2$ iff one of the following conditions is satisfied:

1. $\sigma_1 = (P_1, N_1, D_1) \in bCAN$, $\sigma_2 = (P_2, N_2, D_2) \in bCAN$, $N_1 = N_2$ and $D_1 = D_2$
2. $\sigma_1 = (P, N, D) \in bCAN$, $\sigma_2 = ((\widehat{P}, \widehat{N}, \widehat{D}), \mathbf{v}) \in \Sigma_{\widehat{bCAN},\mathcal{H}}$, $N = \mathsf{age}(\widehat{N}, \mathbf{v})$ and $D = \widehat{D}$
3. as (2) with the roles of σ_1 and σ_2 reversed
4. $\sigma_1 = ((\widehat{P}_1, \widehat{N}_1, \widehat{D}_1), \mathbf{v}_1) \in \Sigma_{\widehat{bCAN},\mathcal{H}}$, $\sigma_2 = ((\widehat{P}_2, \widehat{N}_2, \widehat{D}_2), \mathbf{v}_2) \in \Sigma_{\widehat{bCAN},\mathcal{H}}$, $\mathsf{age}(\widehat{N}_1, \mathbf{v}_1) = \mathsf{age}(\widehat{N}_2, \mathbf{v}_2)$ and $\widehat{D}_1 = \widehat{D}_2$. □

Proposition B.3 *$\approx_{ND}$ is an equivalence relation.*

Proof Immediate from Definition B.4. □

Strong equivalence of both clocked and unclocked *bCANDLE* states is defined simply as $\approx_{ND}$-bisimilarity, and related definitions are obtained in the obvious way.

Remark B.1 Notice that Propositions 3.9 and 3.10 are valid when extended to clocked systems, i.e. $\underline{\leftrightarrow}$ is a congruence for the operators of $\widehat{Proc}$ and the equational laws are sound for $\widehat{bCAN}$.

Now, we turn our attention to addressing a technical point concerning the use of $\checkmark$. We wish to obtain a compositional proof and to avoid the need to reason about the persistence of systems. In order to achieve this, it is convenient to treat $\checkmark$ as a (distinguished) process term and to introduce locations $(\checkmark, \widehat{N}, \widehat{D})$ corresponding to systems $(\checkmark, N, D)$. We silently extend $bCAN$ and $\widehat{bCAN}$ to contain these additional systems. The definition of age is extended by $\mathsf{age}(\checkmark, \mathbf{v}) \mathrel{\widehat{=}} \checkmark$, and the invariant function I by $I(\checkmark, \widehat{N}, \widehat{D}) \mathrel{\widehat{=}} h_u \leq 0 \wedge I(\widehat{N})$. Now, as currently defined, a system $(\checkmark, N, D)$ has no transitions. But if a clock valuation $\mathbf{v}$ satisfies $I(\checkmark, \widehat{N}, \widehat{D})$, then $((\checkmark, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{0} ((\checkmark, \widehat{N}, \widehat{D}), \mathbf{v})$. To resolve this discrepancy, we assume that the semantics of *bCANDLE* is extended with a rule

$$\textbf{P-}\checkmark \ \frac{}{(\checkmark, N, D) \xrightarrow{0} (\checkmark, N, D)}$$

Now it is clear that $((\checkmark, \widehat{N}, \widehat{D}), \mathbf{v}[h_u := 0])$ is strongly bisimilar to $\mathsf{age}((\checkmark, \widehat{N}, \widehat{D}), \mathbf{v}[h_u := 0])$, each state having only a 0-transition to itself.

The following proposition asserts that $\checkmark$ really is a distinguished process term.

Proposition B.4 *For any bCANDLE system $(P, N, D) \in bCAN$,*

if $(P, N, D) \mathrel{\underline{\leftrightarrow}} (\checkmark, N, D)$, then $P \equiv \checkmark$.

Proof A standard induction. Intuitively, we can see from the semantics that $\checkmark$ is the only process which does not allow either the immediate execution of some discrete action or the passage of time by some strictly positive amount. $\square$

We now introduce several lemmas which will be useful in the main proof. Each lemma is introduced by a few words of informal explanation.

The first two lemmas simply assert that network behaviour is both *independently determined* and *non-intrusive* in both clocked and unclocked systems. Network behaviour is independently determined in the sense that each new network state is uniquely determined by a current network state and a network action, irrespective of the system context. It is non-intrusive in that the process and data components of a system state are unchanged by network transitions.

Lemma B.1 *Let $(P_1, N, D_1), (P_1', N', D_1'), (P_2, N, D_2)$ and (P_2', N'', D_2') be bCANDLE system states in bCAN. Let $\lambda \in A_{\mathsf{n}} \cup \mathbb{R}$ be a network transition label. Then, if*

$$(P_1, N, D_1) \xrightarrow{\lambda} (P_1', N', D_1') \text{ and } (P_2, N, D_2) \xrightarrow{\lambda} (P_2', N'', D_2')$$

we have

1. *the new network state is uniquely determined, i.e. $N' = N''$, and*
2. *the data component is unchanged, i.e. $D_1 = D_1'$.*

Additionally, for $\lambda \in A_{\mathsf{n}}$, we have

3. *the process component is unchanged, i.e. $P_1 \equiv P_1'$.*

Proof It is clear from the network rules (Definition 3.15) that a new network state is uniquely determined by the current state and the network action. When included in a system context, the rules for basic systems and data-guarded systems show that the new network state is unaffected by the process and data components, which are themselves unchanged by the network transition in the case of a network action (only the data component being unchanged in the case of a strictly positive delay action). This property is preserved by all the process operators (Definition 3.25). $\square$

Lemma B.2 *Let $\widehat{B} \in b\widehat{CAN}$ be a clocked bCANDLE system and $\mathcal{G}(\widehat{B}) = (Q, q^{\mathcal{I}}, \mathcal{H}, E, I)$ its corresponding TA. Let $(\widehat{P}_1, \widehat{N}, \widehat{D}_1), (\widehat{P}'_1, \widehat{N}', \widehat{D}'_1), (\widehat{P}_2, \widehat{N}, \widehat{D}_2)$ and $(\widehat{P}'_2, \widehat{N}'', \widehat{D}'_2)$ be locations in Q. Let $\lambda_\mathsf{n} \in A_\mathsf{n}$ be a network action label. Then, if E contains edges*

$$(\widehat{P}_1, \widehat{N}, \widehat{D}_1) \xrightarrow{\psi_1, \lambda_\mathsf{n}, \mathsf{H}_1} (\widehat{P}'_1, \widehat{N}', \widehat{D}'_1) \text{ and } (\widehat{P}_2, \widehat{N}, \widehat{D}_2) \xrightarrow{\psi_2, \lambda_\mathsf{n}, \mathsf{H}_2} (\widehat{P}'_2, \widehat{N}'', \widehat{D}'_2)$$

we have

1. *the new network state is uniquely determined, i.e. $\widehat{N}' = \widehat{N}''$,*
2. *the process and data components are unchanged, i.e. $\widehat{P}_1 = \widehat{P}'_1$ and $\widehat{D}_1 = \widehat{D}'_1$, and*
3. *the clock guards and reset sets are identical, i.e. $\psi_1 \equiv \psi_2$ and $\mathsf{H}_1 = \mathsf{H}_2$.*

Proof Similar to Lemma B.1, using Definition 4.15. □

The next three lemmas are concerned with the effect of clock resets on the satisfaction of location invariants.

If a clock valuation satisfies a location invariant, then any clock valuation derived from it by resetting some clocks also satisfies the location invariant.

Lemma B.3 *Let $\mathcal{H}$ be a set of clock variables, $\mathsf{H} \subseteq \mathcal{H}$ and $\mathbf{v} \in \mathbb{R}^{\mathcal{H}}$ a $\mathcal{H}$-valuation. Let $\widehat{B} \in b\widehat{CAN}$ be a clocked bCANDLE system, where $\mathsf{clk}(\widehat{B}) \subseteq \mathcal{H}$. If $\mathbf{v} \models I(\widehat{B})$, then $\mathbf{v}[\mathsf{H} := 0] \models I(\widehat{B})$.*

Proof Induction on the number of steps in the expansion of $I(\widehat{B})$, using Definition 4.16. □

If a clock valuation satisfies the invariant of a process term in *some* data environment, then any clock valuation derived from it by resetting some clocks satisfies the invariant in *any* compatible data environment, provided the urgent clock h_u is reset.

Lemma B.4 *Let $\mathcal{H}$ be a set of clock variables and $\mathbf{v} \in \mathbb{R}^{\mathcal{H}}$ a $\mathcal{H}$-valuation. Let $\widehat{P} \in \widehat{Proc}$ be a clocked process term, where $\mathsf{clk}(\widehat{P}) \subseteq \mathcal{H}$. Let $\mathsf{H} \subseteq \mathcal{H}$ with $h_u \in \mathsf{H}$. Then, if there is some data environment $\widehat{D} \in DataEnv$ such that $\mathbf{v} \models I(\widehat{P}, \widehat{D})$, it is the case that for all $\widehat{D}' \in DataEnv$, such that $\widehat{D}'$ and $\widehat{D}$ are compatible, we have $\mathbf{v}[\mathsf{H} := 0] \models I(\widehat{P}, \widehat{D}')$.*

Proof By induction on the number of steps in the expansion of $I(\widehat{P}, \widehat{D})$, using Definition 4.16. □

If the initial clocks of a process term, together with the urgent clock h_u, are all reset in some clock valuation, then the resulting clock valuation satisfies the process term invariant.

Lemma B.5 *Let $\mathcal{H}$ be a set of clock variables, $\mathbf{v} \in \mathbb{R}^{\mathcal{H}}$ a $\mathcal{H}$-valuation and $\widehat{P} \in \widehat{Proc}$ a clocked term, where $\mathsf{clk}(\widehat{P}) \subseteq \mathcal{H}$. Let $\mathsf{H} \subseteq \mathcal{H}$. Then, if $\mathsf{iclk}(\widehat{P}) \cup \{h_u\} \subseteq \mathsf{H}$, it is the case that $\mathbf{v}[\mathsf{H} := 0] \models I(\widehat{P}, \widehat{D})$, in any data environment $\widehat{D}$.*

Proof By induction on the number of steps in the expansion of $I(\widehat{P}, \widehat{D})$, using Definitions 4.7 and 4.16. $\square$

The remaining lemmas are concerned with properties of the age function.

Only the values of network clocks can affect the result of aging a network, and only the values of initial clocks can affect the result of aging a process term.

Lemma B.6 *Let $\mathcal{H}$ be a set of clock variables and $\mathbf{v}, \mathbf{v}' \in \mathbb{R}^{\mathcal{H}}$ be $\mathcal{H}$-valuations.*

1. *Let $\widehat{N} \in \widehat{Network}$ be a clocked network. Then,*

$$(\mathsf{age}(\widehat{N}, \mathbf{v}) = N \wedge \forall\, h \in \mathsf{clk}(\widehat{N}) \;.\; \mathbf{v}'(h) = \mathbf{v}(h)) \Rightarrow \mathsf{age}(\widehat{N}, \mathbf{v}') = N$$

2. *Let $\widehat{P} \in \widehat{Proc}$ be a clocked process term. Then,*

$$(\mathsf{age}(\widehat{P}, \mathbf{v}) = P \wedge \forall\, h \in \mathsf{iclk}(\widehat{P}) \;.\; \mathbf{v}'(h) = \mathbf{v}(h)) \Rightarrow \mathsf{age}(\widehat{P}, \mathbf{v}') = P$$

Proof Immediate from Definition B.2 for $\widehat{N}$ and by induction on the number of steps in the expansion of $\mathsf{age}(\widehat{P}, \mathbf{v})$ for $\widehat{P}$. $\square$

If the initial clocks of a process term are all reset in some clock valuation, then the aging of the term by that clock valuation produces a term which is equivalent to the corresponding unclocked term.

Lemma B.7 *Let $\mathcal{H}$ be a set of clock variables, $\mathbf{v} \in \mathbb{R}^{\mathcal{H}}$ a $\mathcal{H}$-valuation and $\widehat{P} \in \widehat{Proc}$ a clocked term, where $\mathsf{clk}(\widehat{P}) \subseteq \mathcal{H}$. Let $\mathsf{H} \subseteq \mathcal{H}$. If $\mathsf{iclk}(\widehat{P}) \subseteq \mathsf{H}$, then $\mathsf{age}(\widehat{P}, \mathbf{v}[\mathsf{H} := 0]) \leftrightarrow \mathsf{unclk}(\widehat{P})$.*

Proof By induction on the number of steps in the expansion of $\mathsf{age}(\widehat{P}, \mathbf{v}[\mathsf{H} := 0])$ using Definitions 4.7, 4.8 and B.2. Intuitively, we can see that for all terms, except those containing terms of the form $\mathrm{rec}\, X.\widehat{P}$, age and unclk give results which are syntactically identical. In the case of $\mathrm{rec}\, X.\widehat{P}$, unclk simply removes the clock variables, whereas age unwinds the recursion until there is no leading rec, and then removes the clock variables. In either case, it is clear that the results are equivalent. $\square$

If a clock valuation can be increased by time t, while satisfying the invariant of a clocked network, then the corresponding aged network allows the passage of time t.

Lemma B.8 *Let $\mathcal{H}$ be a set of clock variables. Let K be a finite set of channel identifiers and $\widehat{N} \in \widehat{Network}_K$ a network over K, where $\mathsf{clk}(\widehat{N}) \subseteq \mathcal{H}$. Let $\mathbf{v} \in \mathbb{R}^{\mathcal{H}}$ be a $\mathcal{H}$-valuation and $t \in \mathbb{R}$. Then,*

$$\mathbf{v} + t \models I(\widehat{N}) \Rightarrow t \leq \mathsf{tcp}(\mathsf{age}(\widehat{N}, \mathbf{v}))$$

Proof We assume that $\mathbf{v}+t \models I(\widehat{N})$ and show that $\forall\, k \in K \;.\; t \leq \mathsf{tcp}(\mathsf{age}(\widehat{N}_k, \mathbf{v}))$. The result follows directly from Definition 3.12 and **N.5**. The proof proceeds by case analysis on channel status. We illustrate for the case $\widehat{N}_k = (\overset{t_1,t_2}{\rightsquigarrow} m, u)^h$.

1. **Case:** $\widehat{N}_k = (\stackrel{t_1,t_2}{\rightsquigarrow} m, u)^h$.
   By Definition B.2, $\mathsf{age}(\widehat{N}_k, \mathbf{v}) = (\stackrel{t_1 \dot{-} \mathbf{v}(h), t_2 \dot{-} \mathbf{v}(h)}{\rightsquigarrow} m, u)$. By Definition 3.12, $\mathsf{tcp}(\stackrel{t_1 \dot{-} \mathbf{v}(h), t_2 \dot{-} \mathbf{v}(h)}{\rightsquigarrow} m, u) = t_2 \dot{-} \mathbf{v}(h)$. But, if $\mathbf{v} + t \models I(\widehat{N}_k)$, then $\mathbf{v} + t \models$ $\mathsf{if}\ t_2 \in \mathbb{N}\ \mathsf{then}\ h \leq t_2\ \mathsf{else}\ \mathtt{tt}$, and

$$\begin{array}{rcl} t_2 \in \mathbb{N} & \Rightarrow & \mathbf{v} + t \models h \leq t_2 \\ & \Rightarrow & \mathbf{v}(h) + t \leq t_2, \quad \text{by Definition of } \models \\ & \Rightarrow & t \leq t_2 - \mathbf{v}(h) \\ & \Rightarrow & t \leq \mathsf{tcp}(\mathsf{age}(\widehat{N}_k, \mathbf{v})) \\ t_2 = \infty & \Rightarrow & t_2 \dot{-} \mathbf{v}(h) = \infty \\ & \Rightarrow & t \leq \mathsf{tcp}(\mathsf{age}(\widehat{N}_k, \mathbf{v})) \end{array}$$

The other cases are similar. □

Main Proof

It is now possible to state the proof of Proposition B.1.

Let $\mathcal{T}[\![\mathcal{G}^+(\widehat{B})]\!] = (\Sigma_1, \sigma_1^{\mathcal{I}}, L_1, \longrightarrow_1)$ and $\mathcal{T}[\![B]\!] = (\Sigma_2, \sigma_2^{\mathcal{I}}, L_2, \longrightarrow_2)$. We show that the relation R is a $\approx_{ND}$-bisimulation up to $\underline{\leftrightarrow}$, where

$$\mathsf{R} \mathrel{\widehat{=}} \{((\widehat{B}, \mathbf{v}), \mathsf{age}(\widehat{B}, \mathbf{v})) \mid (\widehat{B}, \mathbf{v}) \in \Sigma_{b\widehat{CAN},\mathcal{H}}\}$$

and $\sigma_1^{\mathcal{I}} \underline{\leftrightarrow} \mathsf{R} \underline{\leftrightarrow} \sigma_2^{\mathcal{I}}$. The proof of the proposition follows from Remark 4.2 and the transitivity of $\underline{\leftrightarrow}$.

To show that R is a $\approx_{ND}$-bisimulation up to $\underline{\leftrightarrow}$, we reason as follows. Let $\widehat{\sigma} \mathsf{R} \sigma$.

1. It is clear from Definitions B.2, B.4 and the definition of R, that $\widehat{\sigma} \approx_{ND} \sigma$.

2. It is enough to show that for all $\lambda \in L$, if $\sigma = (P, N, D) \xrightarrow{\lambda} (P', N', D')$, then $\widehat{\sigma} = ((\widehat{P}, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{\lambda} ((\widehat{P}', \widehat{N}', \widehat{D}'), \mathbf{v}')$, and there exist $\widehat{P}'' \underline{\leftrightarrow} \widehat{P}'$ and $P'' \underline{\leftrightarrow} P'$ such that $((\widehat{P}'', \widehat{N}', \widehat{D}'), \mathbf{v}') \mathsf{R} (P'', N', D')$. The proof is by induction on the number of steps in the calculation of $\mathsf{age}(\widehat{\sigma})$. We proceed by case analysis on the structure of $\widehat{\sigma}$.

   (a) **Case:** $\widehat{\sigma} \equiv ((k!i.x, \widehat{N}, \widehat{D}), \mathbf{v})$.
   So $\sigma \equiv (k!i.x, N, D)$, where $\mathsf{age}(\widehat{N}, \mathbf{v}) = N$ and $\widehat{D} = D$.

   There are three sub-cases to consider:

   i. **Snd.1**: $\lambda = k!i.v$, $\sigma' \equiv (\checkmark, N', D)$, where $N_k = (s, u)$, $v = D.x$, and $N' = N[k := (s, u \hookleftarrow i.v)]$.
   In this case, since $\mathsf{age}(\widehat{N}, \mathbf{v}) = N$, then $\widehat{N}_k = (\hat{s}, \hat{u})^h$, where $\mathsf{age}(\hat{s}, \hat{u})^h, \mathbf{v}) = (s, u)$ which implies, by Definition B.2, that $u = \hat{u}$. And, since $\widehat{D} = D$, then $\widehat{D}.x = v$. So by **E_Snd.1**, there is an edge $(k!i.x, \widehat{N}, \widehat{D}) \stackrel{\mathtt{tt}, k!i.v, \{h_u\}}{\longrightarrow} (\checkmark, \widehat{N}', \widehat{D})$, where $\widehat{N}' =$

$\widehat{N}[k := (\hat{s}, u \hookleftarrow i.v)^h]$. Clearly, $\mathbf{v} \models \mathtt{tt}$ and, since $\mathbf{v} \models I(\widehat{N})$, then, by Definition 4.16, $\mathbf{v}[h_u := 0] \models I(\widehat{N}')$, so by **TA.1**, there is a transition $((k!i.x, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{k!i.v} ((\checkmark, \widehat{N}', \widehat{D}), \mathbf{v}[h_u := 0])$. It is easy to see that $\mathsf{age}(\widehat{N}', \mathbf{v}[h_u := 0]) = N'$, and, therefore, $((\checkmark, \widehat{N}', \widehat{D}), \mathbf{v}[h_u := 0]) \mathsf{R} (\checkmark, N', D)$.

ii. **Snd.2**: $\lambda \in A_{\mathsf{n}}$, $\sigma' \equiv (k!i.x, N', D)$.
There are four sub-cases to consider: one for each of the ways in which the network transition can be derived.

A. **N.1**: similar to the following case.

B. **N.2**: $\lambda = k\!\uparrow m$, $N_k = (\overset{0,t}{\rightsquigarrow} m, u)$, $N' = N[k := (\uparrow m, u)]$
In this case, since $\mathsf{age}(\widehat{N}, \mathbf{v}) = N$, then, by Definition B.2, $\widehat{N}_k = (\overset{t_1,t_2}{\rightsquigarrow} m, u)^h$ and $\mathbf{v}(h) \geq t_1$. So by **E_Snd.2** and **E_N.2**, there is an edge $(k!i.x, \widehat{N}, \widehat{D}) \xrightarrow{h \geq t_1, k\uparrow m, \{h_u\}} (k!i.x, \widehat{N}', \widehat{D})$, where $\widehat{N}' = \widehat{N}[k := (\uparrow m, u)^h]$. Clearly, $\mathbf{v} \models h \geq t_1$, and $\mathbf{v}[h_u := 0] \models I(k!i.x, \widehat{N}', \widehat{D})$, so by **TA.1**, there is a transition $((k!i.x, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{k\uparrow m} ((k!i.x, \widehat{N}', \widehat{D}), \mathbf{v}[h_u := 0])$. Obviously, $((k!i.x, \widehat{N}', \widehat{D}), \mathbf{v}[h_u := 0]) \mathsf{R} (k!i.x, N', D)$.

C. **N.3**: similar to the previous case.

D. **N.4**: similar to the previous case.

iii. **Snd.3**: $\lambda = 0$, $\sigma' \equiv (k!i.x, N, D)$.
Since $\mathbf{v} \models I(k!i.x, \widehat{N}, \widehat{D})$, then by **TA.2**, $((k!i.x, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{0} ((k!i.x, \widehat{N}, \widehat{D}), \mathbf{v})$. We already have that $((k!i.x, \widehat{N}, \widehat{D}), \mathbf{v}) \mathsf{R} (k!i.x, N, D)$.

(b) **Case:** $\widehat{\sigma} \equiv ((k?i.x, \widehat{N}, \widehat{D}), \mathbf{v})$: similar to the previous case.

(c) **Case:** $\widehat{\sigma} \equiv (([\omega : t_1, t_2]^h, \widehat{N}, \widehat{D}), \mathbf{v})$: similar to the previous case.

(d) **Case:** $\widehat{\sigma} \equiv ((\gamma \to P, \widehat{N}, \widehat{D}), \mathbf{v})$: similar to the following case.

(e) **Case:** $\widehat{\sigma} \equiv ((\widehat{P} \,;\, \widehat{Q}, \widehat{N}, \widehat{D}), \mathbf{v})$
So, since $\widehat{\sigma} \mathsf{R} \sigma$, we have $\sigma \equiv (P \,;\, Q, N, D)$, where $P = \mathsf{age}(\widehat{P}, \mathbf{v})$, $Q = \mathsf{unclk}(\widehat{Q})$, $N = \mathsf{age}(\widehat{N}, \mathbf{v})$, and $D = \widehat{D}$.
There are two sub-cases to consider:

i. **Seq.1**: $\lambda \in A_{\mathsf{p}} \cup A_{\mathsf{n}} \cup \mathbb{R}$, $\sigma' \equiv (P' \,;\, Q, N', D')$, $P' \not\equiv \checkmark$
By **Seq.1**, $(P, N, D) \xrightarrow{\lambda} (P', N', D')$, where $P' \not\equiv \checkmark$, and so, by i.h., $((\widehat{P}, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{\lambda} ((\widehat{P}', \widehat{N}', \widehat{D}'), \mathbf{v}')$, and there exist $\widehat{P}'' \underline{\leftrightarrow} \widehat{P}'$ and $P'' \underline{\leftrightarrow} P'$ such that $((\widehat{P}'', \widehat{N}', \widehat{D}'), \mathbf{v}') \mathsf{R} (P'', N', D')$. There are now two sub-cases to consider:

A. $\lambda \in A_{\mathsf{p}} \cup A_{\mathsf{n}}$: So $((\widehat{P}, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{\lambda} ((\widehat{P}', \widehat{N}', \widehat{D}'), \mathbf{v}')$ must be derived by **TA.1** from an edge $(\widehat{P}, \widehat{N}, \widehat{D}) \xrightarrow{\psi, \lambda, \mathsf{H}} (\widehat{P}', \widehat{N}', \widehat{D}')$, where $\mathbf{v} \models \psi$ and $\mathbf{v}' = \mathbf{v}[\mathsf{H} := 0] \models I(\widehat{P}', \widehat{N}', \widehat{D}')$. Now, by **E_Seq.1**, there must be an edge $(\widehat{P} \,;\, \widehat{Q}, \widehat{N}, \widehat{D}) \xrightarrow{\psi, \lambda, \mathsf{H}} (\widehat{P}' \,;\, \widehat{Q}, \widehat{N}', \widehat{D}')$. We already have $\mathbf{v} \models \psi$ and, since $\mathbf{v}[\mathsf{H} := 0] \models (\widehat{P}', \widehat{N}', \widehat{D}')$, we also have $\mathbf{v}[\mathsf{H} := 0] \models I(\widehat{P}' \,;\, \widehat{Q}, \widehat{N}', \widehat{D}')$, by Definition 4.16. So, by TA.1, there is a transition $((\widehat{P} \,;\, \widehat{Q}, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{\lambda} ((\widehat{P}'; \widehat{Q}, \widehat{N}' \widehat{D}'), \mathbf{v}')$. Since $((\widehat{P}'', \widehat{N}', \widehat{D}'), \mathbf{v}') \mathsf{R}$

(P'', N', D'), and $\mathsf{unclk}(\widehat{Q}) = Q$, then $((\widehat{P}'' ; \widehat{Q}, \widehat{N}', \widehat{D}'), \mathbf{v}')\mathsf{R}$ $(P'' ; Q, N', D')$. Moreover, $\widehat{P}'' ; \widehat{Q} \underline{\leftrightarrow} \widehat{P}' ; \widehat{Q}$ and $P'' ; Q \underline{\leftrightarrow}$ $P' ; Q$, as required.

B. $\lambda = t \in \mathbb{R}$:
So $((\widehat{P}, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{t} ((\widehat{P}', \widehat{N}', \widehat{D}'), \mathbf{v}')$ must be derived by **TA.2** with $(\widehat{P}', \widehat{N}', \widehat{D}') \equiv (\widehat{P}, \widehat{N}, \widehat{D})$, $\mathbf{v}' = \mathbf{v} + t$, and $\forall\, t' \in \mathbb{R} \mid t' \leq t \,.\, \mathbf{v} + t' \models I(\widehat{P}, \widehat{N}, \widehat{D})$. By Definition 4.16, we have $\forall\, t' \in \mathbb{R} \mid t' \leq t \,.\, \mathbf{v} + t' \models I(\widehat{P} ; \widehat{Q}, \widehat{N}, \widehat{D})$, and so by **TA.2**, there is a transition $((\widehat{P}; \widehat{Q}, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{t} ((\widehat{P}'; \widehat{Q}, \widehat{N}, \widehat{D}), \mathbf{v}')$. From i.h., we have $\widehat{P}'' \underline{\leftrightarrow} \widehat{P}'$ and $P'' \underline{\leftrightarrow} P'$ such that $((\widehat{P}'', \widehat{N}, \widehat{D}), \mathbf{v}')\mathsf{R}$ (P'', N', D'), and so, since $\mathsf{unclk}(\widehat{Q}) = Q$, we have $((\widehat{P}'' ;$ $\widehat{Q}, \widehat{N}, \widehat{D}), \mathbf{v}')\mathsf{R}(P'' ; Q, N', D')$, where $\widehat{P}'' ; \widehat{Q} \underline{\leftrightarrow} \widehat{P}' ; \widehat{Q}$ and $P'' ; Q \underline{\leftrightarrow} P' ; Q$.

ii. **Seq.2**: $\lambda = \lambda_{\mathsf{p}} \in A_{\mathsf{p}}$, $\sigma' \equiv (Q, N', D')$
By **Seq.2**, $(P, N, D) \xrightarrow{\lambda_{\mathsf{p}}} (\checkmark, N', D')$, and so, by i.h., $((\widehat{P}, \widehat{N}, \widehat{D}), \mathbf{v})$ $\xrightarrow{\lambda_{\mathsf{p}}} ((\widehat{P}', \widehat{N}', \widehat{D}'), \mathbf{v}')$, where there exist $\widehat{P}'' \underline{\leftrightarrow} \widehat{P}'$ and $P'' \underline{\leftrightarrow} \checkmark$, such that $((\widehat{P}'', \widehat{N}', \widehat{D}'), \mathbf{v}')\mathsf{R}(P'', N', D')$. But, by definition of age and Proposition B.4, we must therefore have $\widehat{P}'' \equiv \widehat{P}' \equiv$ $P'' \equiv \checkmark$.
The transition $((\widehat{P}, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{\lambda_{\mathsf{p}}} ((\checkmark, \widehat{N}', \widehat{D}'), \mathbf{v}')$ must be derived by **TA.1** from an edge $(\widehat{P}, \widehat{N}, \widehat{D}) \xrightarrow{\psi, \lambda_{\mathsf{p}}, \mathsf{H}} (\checkmark, \widehat{N}', \widehat{D}')$, where $\mathbf{v} \models \psi$ and $\mathbf{v}' = \mathbf{v}[\mathsf{H} := 0] \models I(\checkmark, \widehat{N}', \widehat{D}')$. So, by **E_Seq.2**, there is an edge $(\widehat{P} ; \widehat{Q}, \widehat{N}, \widehat{D}) \xrightarrow{\psi, \lambda_{\mathsf{p}}, \mathsf{H} \cup \mathsf{iclk}(\widehat{Q})} (\widehat{Q}, \widehat{N}', \widehat{D}')$. We already have $\mathbf{v} \models \psi$ and, by Lemmas B.3 and B.5, we have $\mathbf{v}[\mathsf{H} \cup \mathsf{iclk}(\widehat{Q}) := 0] \models (\widehat{Q}, \widehat{N}', \widehat{D}')$. So, by TA.1, there is a transition $((\widehat{P} ; \widehat{Q}, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{\lambda_{\mathsf{p}}} ((\widehat{Q}, \widehat{N}', \widehat{D}'), \mathbf{v}'')$, where $\mathbf{v}'' = \mathbf{v}[\mathsf{H} \cup$ $\mathsf{iclk}(\widehat{Q}) := 0]$. Since $((\checkmark, \widehat{N}', \widehat{D}'), \mathbf{v}')\mathsf{R}(\checkmark, N', D')$, then, $D' =$ $\widehat{D}'$, and, by Lemma B.6, $N' = \mathsf{age}(\widehat{N}', \mathbf{v}'')$. Let $\widetilde{Q} = \mathsf{age}(\widehat{Q}, \mathbf{v}'')$. Clearly, $((\widehat{Q}, \widehat{N}', \widehat{D}'), \mathbf{v}'')\mathsf{R}(\widetilde{Q}, N', D')$. But, $Q = \mathsf{unclk}(\widehat{Q})$, and by Lemma B.7, $\mathsf{unclk}(\widehat{Q}) \underline{\leftrightarrow} \widetilde{Q}$, so $Q \underline{\leftrightarrow} \widetilde{Q}$, as required.

(f) **Case:** $\widehat{\sigma} \equiv ((\widehat{P} + \widehat{Q}, \widehat{N}, \widehat{D}), \mathbf{v})$: similar to the following case.

(g) **Case:** $\widehat{\sigma} \equiv ((\widehat{P} \;[\!> \widehat{Q}, \widehat{N}, \widehat{D}), \mathbf{v})$.
So, since $\widehat{\sigma}\mathsf{R}\sigma$, we have $\sigma \equiv (P \;[\!> Q, N, D)$, where $\mathsf{age}(\widehat{P}, \mathbf{v}) = P$, $\mathsf{age}(\widehat{Q}, \mathbf{v}) = Q$, $\mathsf{age}(\widehat{N}, \mathbf{v}) = N$ and $\widehat{D} = D$.
There are four sub-cases to consider:

i. **Int.1**: $\lambda = \lambda_{\mathsf{p}} \in A_{\mathsf{p}}$, $\sigma' \equiv (P' \;[\!> Q, N', D')$, $P' \not\equiv \checkmark$.
By **Int.1**, $(P, N, D) \xrightarrow{\lambda_{\mathsf{p}}} (P', N', D')$ and so, by i.h., $((\widehat{P}, \widehat{N}, \widehat{D}), \mathbf{v})$ $\xrightarrow{\lambda_{\mathsf{p}}} ((\widehat{P}', \widehat{N}', \widehat{D}'), \mathbf{v}')$, and there exist $\widehat{P}'' \underline{\leftrightarrow} \widehat{P}'$ and $P'' \underline{\leftrightarrow} P'$ such that $((\widehat{P}'', \widehat{N}', \widehat{D}'), \mathbf{v}')\mathsf{R}(P'', N', D')$. But this transition must be derived by **TA.1** from an edge $(\widehat{P}, \widehat{N}, \widehat{D}) \xrightarrow{\psi, \lambda_{\mathsf{p}}, \mathsf{H}} (\widehat{P}', \widehat{N}', \widehat{D}')$, where $\widehat{P}' \not\equiv \checkmark$, $\mathbf{v} \models \psi$ and $\mathbf{v}' = \mathbf{v}[\mathsf{H} := 0] \models I(\widehat{P}', \widehat{N}', \widehat{D}')$. So, by **E_Int.1**, there is an edge $(\widehat{P} \;[\!> \widehat{Q}, \widehat{N}, \widehat{D}) \xrightarrow{\psi, \lambda_{\mathsf{p}}, \mathsf{H}} (\widehat{P}' \;[\!>$

$\widehat{Q}, \widehat{N}', \widehat{D}')$. We already have $\mathbf{v}[\mathsf{H} := 0] \models I(\widehat{P}', \widehat{N}', \widehat{D}')$ and $\mathbf{v} \models I(\widehat{Q}, \widehat{N}, \widehat{D})$. Since the urgent clock h_u is reset by every edge, then, by Lemmas B.3 and B.4, we have $\mathbf{v}[\mathsf{H} := 0] \models I(\widehat{Q}, \widehat{D}')$. So, by Definition 4.16, we have $\mathbf{v}[\mathsf{H} := 0] \models I(\widehat{P}' \; [> \widehat{Q}, \widehat{N}', \widehat{D}')$. Moreover, $\mathbf{v} \models \psi$, so by **TA.1**, there is a transition $((\widehat{P} \; [> \widehat{Q}, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{\lambda_\mathsf{p}} ((\widehat{P}' \; [> \widehat{Q}, \widehat{N}', \widehat{D}'), \mathbf{v}[\mathsf{H} := 0])$.
Since $\widehat{P}'' \leftrightarroweq \widehat{P}'$ and $P'' \leftrightarroweq P'$, then $\widehat{P}'' \; [> \widehat{Q} \leftrightarroweq \widehat{P}' \; [> \widehat{Q}$ and $P'' \; [> Q \leftrightarroweq P' \; [> Q$. To see that $((\widehat{P}'' \; [> \widehat{Q}, \widehat{N}', \widehat{D}'), \mathbf{v}[\mathsf{H} := 0])\mathsf{R}(P'' \; [> Q, N', D')$, we reason as follows. By construction of R, $P'' = \mathsf{age}(\widehat{P}'', \mathbf{v}[\mathsf{H} := 0])$. Furthermore, by safety of clock variable allocation, $\mathsf{H} \cap \mathsf{iclk}(\widehat{Q}) = \emptyset$, so, by Lemma B.6, $Q = \mathsf{age}(\widehat{Q}, \mathbf{v}[\mathsf{H} := 0])$ and $N' = \mathsf{age}(\widehat{N}', \mathbf{v}[\mathsf{H} := 0])$. The result follows by construction of R.

ii. **Int.2**: similar to previous case.

iii. **Int.3**: similar to previous case.

iv. **Int.4**: $\lambda = \lambda_\mathsf{nt} \in A_\mathsf{n} \cup \mathbb{R}$, $\sigma' \equiv (P' \; [> Q', N', D')$
There are two sub-cases to consider:

A. $\lambda = \lambda_\mathsf{n} \in A_\mathsf{n}$:
In this case, by Lemma B.1, we have $P' \; [> Q' \equiv P \; [> Q$ and $D' = D$. So, by **Int.4**, $(P, N, D) \xrightarrow{\lambda_\mathsf{n}} (P, N', D)$ and $(Q, N, D) \xrightarrow{\lambda_\mathsf{n}} (Q, N', D)$. Moreover, by i.h. and Lemma B.2, $((\widehat{P}, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{\lambda_\mathsf{n}} ((\widehat{P}, \widehat{N}', \widehat{D}), \mathbf{v}')$ and $(\widehat{Q}, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{\lambda_\mathsf{n}} ((\widehat{Q}, \widehat{N}', \widehat{D}), \mathbf{v}')$, and there exist $\widehat{P}'' \leftrightarroweq \widehat{P}$, $\widehat{Q}'' \leftrightarroweq \widehat{Q}$, $P'' \leftrightarroweq P$ and $Q'' \leftrightarroweq Q$ such that $((\widehat{P}'', \widehat{N}', \widehat{D}), \mathbf{v}')\mathsf{R}(P'', N', D)$ and $((\widehat{Q}'', \widehat{N}', \widehat{D}), \mathbf{v}')\mathsf{R}(Q'', N', D)$.
But $((\widehat{P}, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{\lambda_\mathsf{n}} ((\widehat{P}, \widehat{N}', \widehat{D}), \mathbf{v}')$, must be derived by **TA.1** from an edge $(\widehat{P}, \widehat{N}, \widehat{D}) \xrightarrow{\psi, \lambda_\mathsf{n}; \mathsf{H}} (\widehat{P}, \widehat{N}', \widehat{D})$, where $\mathbf{v} \models \psi$ and $\mathbf{v}' = \mathbf{v}[\mathsf{H} := 0] \models I(\widehat{P}, \widehat{N}', \widehat{D})$. Similarly for $((\widehat{Q}, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{\lambda_\mathsf{n}} ((\widehat{Q}, \widehat{N}', \widehat{D}), \mathbf{v}')$ — Lemma B.2 ensures that this edge will have the same clock guard and reset set as the edge for $(\widehat{P}, \widehat{N}, \widehat{D})$. So by **E_Int.4**, there is an edge $(\widehat{P}[>\widehat{Q}, \widehat{N}, \widehat{D}) \xrightarrow{\psi, \lambda_\mathsf{n}, \mathsf{H}} (\widehat{P}[>\widehat{Q}, \widehat{N}', \widehat{D})$, and, since $\mathbf{v} \models \psi$ and $\mathbf{v}' \models I(\widehat{P}[>\widehat{Q}, \widehat{N}', \widehat{D})$, there is a transition $((\widehat{P}[>\widehat{Q}, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{\lambda_\mathsf{n}} ((\widehat{P}[>\widehat{Q}, \widehat{N}', \widehat{D}), \mathbf{v}')$. Clearly, by i.h. and Definition B.2, $((\widehat{P}''[>\widehat{Q}'', \widehat{N}', \widehat{D}), \mathbf{v}')\mathsf{R}(P''[> Q'', N', D)$, where $\widehat{P}'' \; [> \widehat{Q}'' \leftrightarroweq \widehat{P} \; [> \widehat{Q}$ and $P'' \; [> Q'' \leftrightarroweq P \; [> Q$.

B. $\lambda = t \in \mathbb{R}$:
In this case, by Lemma B.1, we have $D' = D$. So, by **Int.4**, $(P, N, D) \xrightarrow{t} (P', N', D)$ and $(Q, N, D) \xrightarrow{t} (Q', N', D)$. Moreover, by i.h., $((\widehat{P}, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{t} ((\widehat{P}', \widehat{N}', \widehat{D}'), \mathbf{v}')$ and $(\widehat{Q}, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{t} ((\widehat{Q}', \widehat{N}'', \widehat{D}''), \mathbf{v}'')$, and there exist $\widehat{P}'' \leftrightarroweq \widehat{P}$, $\widehat{Q}'' \leftrightarroweq \widehat{Q}$, $P'' \leftrightarroweq P$ and $Q'' \leftrightarroweq Q$ such that $((\widehat{P}'', \widehat{N}', \widehat{D}'), \mathbf{v}')\mathsf{R}(P'', N', D)$ and $((\widehat{Q}'', \widehat{N}'', \widehat{D}''), \mathbf{v}'')\mathsf{R}(Q'', N', D)$. But, $((\widehat{P}, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{t}$

$((\widehat{P}, \widehat{N}, \widehat{D}), \mathbf{v}')$, must be derived by **TA.2**, where $(\widehat{P}', \widehat{N}', \widehat{D}') \equiv (\widehat{P}, \widehat{N}, \widehat{D})$, $\mathbf{v}' = \mathbf{v} + t$ and $\forall\, t' \in \mathbb{R} \mid t' \leq t \,.\, \mathbf{v} + t' \models I(\widehat{P}, \widehat{N}, \widehat{D})$. Similarly for $\widehat{Q}$. So, from Definition 4.16, it is clear that $\forall\, t' \in \mathbb{R} \mid t' \leq t \,.\, \mathbf{v} + t' \models I(\widehat{P} \,[>\, \widehat{Q}, \widehat{N}, \widehat{D})$, and, therefore, by **TA.2**, $((\widehat{P}[>\widehat{Q}, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{t} ((\widehat{P}[>\widehat{Q}, \widehat{N}, \widehat{D}), \mathbf{v}+t)$. By i.h and construction of R, $(\widehat{P}'' \,[>\, \widehat{Q}'', \widehat{N}, \widehat{D}), \mathbf{v} + t)\mathsf{R}(P'' \,[>\, Q'', N', D)$, where $\widehat{P}'' \,[>\, \widehat{Q}'' \underline{\leftrightarrow} \widehat{P} \,[>\, \widehat{Q}$ and $P'' \,[>\, Q'' \underline{\leftrightarrow} P' \,[>\, Q'$.

(h) **Case:** $\widehat{\sigma} \equiv ((\widehat{P} \mid \widehat{Q}, \widehat{N}, \widehat{D}), \mathbf{v})$: similar to the previous case.

(i) **Case:** $\widehat{\sigma} \equiv ((\mathrm{rec}\, X . \widehat{P}_1, \widehat{N}, \widehat{D}), \mathbf{v})$
So, since $\widehat{\sigma}\mathsf{R}\sigma$, we have $\sigma \equiv (P, N, D)$, where $P = \mathsf{age}(\mathrm{rec}\, X . \widehat{P}_1, \mathbf{v})$, $N = \mathsf{age}(\widehat{N}, \mathbf{v})$, and $D = \widehat{D}$. But $\mathsf{age}(\mathrm{rec}\, X . \widehat{P}_1, \mathbf{v}) = \mathsf{age}(\widehat{P}_1[\mathrm{rec}\, X . \widehat{P}_1/X], \mathbf{v})$, which is derived by a shorter calculation, and so, if $(P, N, D) \xrightarrow{\lambda} (P', N', D')$, then, by i.h., $((\widehat{P}_1[\mathrm{rec}\, X . \widehat{P}_1/X], \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{\lambda} ((\widehat{P}', \widehat{N}', \widehat{D}'), \mathbf{v}')$ and there exist $\widehat{P}'' \underline{\leftrightarrow} \widehat{P}'$ and $P'' \underline{\leftrightarrow} P'$ such that $((\widehat{P}'', \widehat{N}', \widehat{D}'), \mathbf{v}')\mathsf{R}(P'', N', D')$. There are two sub-cases to consider:

i. $\lambda \in A_{\mathsf{p}} \cup A_{\mathsf{n}}$:
So $((\widehat{P}_1[\mathrm{rec}\, X . \widehat{P}_1/X], \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{\lambda} ((\widehat{P}', \widehat{N}', \widehat{D}'), \mathbf{v}')$ must be derived by **TA.1** from an edge $(\widehat{P}_1[\mathrm{rec}\, X . \widehat{P}_1/X], \widehat{N}, \widehat{D}) \xrightarrow{\psi,\lambda,\mathsf{H}} (\widehat{P}', \widehat{N}', \widehat{D}')$, where $\mathbf{v} \models \psi$ and $\mathbf{v}' = \mathbf{v}[\mathsf{H} := 0] \models I(\widehat{P}', \widehat{N}', \widehat{D}')$. But then, by **E_Rec**, there is an edge $(\mathrm{rec}\, X . \widehat{P}_1, \widehat{N}, \widehat{D}) \xrightarrow{\psi,\lambda,\mathsf{H}} (\widehat{P}', \widehat{N}', \widehat{D}')$, and so, by **TA.1**, there is a transition $((\mathrm{rec}\, X . \widehat{P}_1, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{\lambda} ((\widehat{P}', \widehat{N}', \widehat{D}'), \mathbf{v}')$. By i.h., we have $((\widehat{P}'', \widehat{N}', \widehat{D}'), \mathbf{v}')\mathsf{R}(P'', N', D')$, where $\widehat{P}'' \underline{\leftrightarrow} \widehat{P}'$ and $P'' \underline{\leftrightarrow} P'$.

ii. $\lambda = t \in \mathbb{R}$:
So $((\widehat{P}_1[\mathrm{rec}\, X . \widehat{P}_1/X], \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{t} ((\widehat{P}', \widehat{N}', \widehat{D}'), \mathbf{v}')$ must be derived by **TA.2**, where $(\widehat{P}', \widehat{N}', \widehat{D}') \equiv (\widehat{P}_1[\mathrm{rec}\, X . \widehat{P}_1/X], \widehat{N}, \widehat{D})$, $\mathbf{v}' = \mathbf{v}+t$ and $\forall\, t' \in \mathbb{R} \mid t' \leq t \,.\, \mathbf{v}+t' \models I(\widehat{P}', \widehat{N}', \widehat{D}')$. But then, by Definition 4.16, $\forall\, t' \in \mathbb{R} \mid t' \leq t \,.\, \mathbf{v}+t' \models I(\mathrm{rec}\, X . \widehat{P}_1, \widehat{N}', \widehat{D}')$, and so, by **TA.2**, there is a transition $((\mathrm{rec}\, X . \widehat{P}_1, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{t} ((\mathrm{rec}\, X . \widehat{P}_1, \widehat{N}', \widehat{D}'), \mathbf{v}')$. By i.h., we have $((\widehat{P}'', \widehat{N}', \widehat{D}'), \mathbf{v}')\mathsf{R}$ (P'', N', D'), where $\widehat{P}'' \underline{\leftrightarrow} \widehat{P}' \equiv \widehat{P}_1[\mathrm{rec}\, X . \widehat{P}_1/X] \underline{\leftrightarrow} \mathrm{rec}\, X . \widehat{P}_1$, and $P'' \underline{\leftrightarrow} P'$, as required.

3. It is enough to show that for all $\lambda \in L$, if $\widehat{\sigma} = ((\widehat{P}, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{\lambda} ((\widehat{P}', \widehat{N}', \widehat{D}'), \mathbf{v}')$, then $\sigma = (P, N, D) \xrightarrow{\lambda} (P', N', D')$, and there exist $\widehat{P}'' \underline{\leftrightarrow} \widehat{P}'$ and $P'' \underline{\leftrightarrow} P'$ such that $((\widehat{P}'', \widehat{N}', \widehat{D}'), \mathbf{v}')\mathsf{R}(P'', N', D')$. Again, the proof is by induction on the number of steps in the calculation of $\mathsf{age}(\widehat{\sigma})$. The proof is symmetrical to the previous case. We provide some illustrative variations.

(a) **Case:** $\widehat{\sigma} \equiv (([\omega : t_1, t_2]^h, \widehat{N}, \widehat{D}), \mathbf{v})$
Since $\widehat{\sigma}\mathsf{R}\sigma$, then $\sigma \equiv (P, N, D)$, where $P = \mathsf{age}(\widehat{P}, \mathbf{v})$, $N = \mathsf{age}(\widehat{N}, \mathbf{v})$ and $D = \widehat{D}$. There are three sub-cases to consider:

i. $\lambda = \omega \in A_{\mathsf{p}}$
A transition $(([\omega : t_1, t_2]^h, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{\omega} ((\widehat{P}', \widehat{N}', \widehat{D}'), \mathbf{v}')$ must be derived by **TA.1** from an edge $([\omega : t_1, t_2]^h, \widehat{N}, \widehat{D}) \xrightarrow{\psi,\omega,\mathsf{H}} (\widehat{P}', \widehat{N}', \widehat{D}')$, where $\mathbf{v} \models \psi$ and $\mathbf{v}' = \mathbf{v}[\mathsf{H} := 0] \models I(\widehat{P}', \widehat{N}', \widehat{D}')$. Such an edge can be constructed only by **E_Comp.1**, so $t_1 \in \mathbb{N}$, $\psi \equiv h \geq t_1$, $\mathsf{H} = \{h_u\}$, $\widehat{P}' = \checkmark$, $\widehat{N}' = \widehat{N}$ and $\widehat{D} \xrightarrow{\omega}_{\mathsf{d}} \widehat{D}'$. Since $\mathbf{v} \models h \geq t_1$ and $\mathbf{v} \models h \leq t_2$, then $t_2 \geq \mathbf{v}(h) \geq t_1$ and so $P = \mathsf{age}([\omega : t_1, t_2]^h, \mathbf{v}) = [\omega : 0, t_2 \dot{-} \mathbf{v}(h)]$. Also, $D = \widehat{D} \xrightarrow{\omega}_{\mathsf{d}} \widehat{D}'$. So by **Comp.1**, $(P, N, D) \xrightarrow{\omega} (\checkmark, N, D')$ where $D' = \widehat{D}'$. Clearly, $((\checkmark, \widehat{N}, \widehat{D}'), \mathbf{v}') \mathsf{R} (\checkmark, N, D')$.

ii. $\lambda = \lambda_{\mathsf{n}} \in A_{\mathsf{n}}$
A transition $(([\omega : t_1, t_2]^h, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{\lambda_{\mathsf{n}}} ((\widehat{P}', \widehat{N}', \widehat{D}'), \mathbf{v}')$ must be derived by **TA.1** from an edge $([\omega : t_1, t_2]^h, \widehat{N}, \widehat{D}) \xrightarrow{\psi,\lambda_{\mathsf{n}},\mathsf{H}} (\widehat{P}', \widehat{N}', \widehat{D}')$, where $\mathbf{v} \models \psi$ and $\mathbf{v}' = \mathbf{v}[\mathsf{H} := 0] \models I(\widehat{P}', \widehat{N}', \widehat{D}')$. Such an edge can be constructed only by **E_Comp.2**, so $\widehat{P}' = \widehat{P}$, $\widehat{D}' = \widehat{D}$ and $\widehat{N} \xrightarrow{\lambda_{\mathsf{n}}}_{\mathsf{n}} \widehat{N}'$. There are four sub-cases to consider: one for each of the ways in which $\widehat{N} \xrightarrow{\lambda_{\mathsf{n}}}_{\mathsf{n}} \widehat{N}'$ can be derived. We show the case **E_N.4**. Cases **E_N.1** – **E_N.3** can be proved similarly.

A. **E_N.4**.
If $\widehat{N} \xrightarrow{\lambda_{\mathsf{n}}}_{\mathsf{n}} \widehat{N}'$ is derived by **E_N.4**, then, for some channel identifier $k \in K$, $\widehat{N}_k = (m \overset{t^{\mathsf{lb}}, t^{\mathsf{ub}}}{\rightsquigarrow}, u)^{h_k}$, $\widehat{N}' = \widehat{N}[k := (\Downarrow, u)^{h_k}]$, $\psi \equiv h_k \geq t^{\mathsf{lb}}$, $\lambda_{\mathsf{n}} = k\Downarrow$ and $\mathsf{H} = \{h_u\}$. We have $\mathbf{v} \models h_k \leq t^{\mathsf{ub}}$ and $\mathbf{v} \models h_k \geq t^{\mathsf{lb}}$, so $t^{\mathsf{lb}} \leq \mathbf{v}(h_k) \leq t^{\mathsf{ub}}$, and therefore $N_k = \mathsf{age}(\widehat{N}_k, \mathbf{v}) = (m \overset{0, t^{\mathsf{ub}} \dot{-} \mathbf{v}(h_k)}{\rightsquigarrow}, u)$. It follows, by **N.4**, that $N \xrightarrow{k\Downarrow}_{\mathsf{n}} N'$, where $N' = N[k := (\Downarrow, u)]$. So, by **Comp.2**, we have $(P, N, D) \xrightarrow{k\Downarrow} (P, N', D)$. To show that $((\widehat{P}, \widehat{N}', \widehat{D}), \mathbf{v}') \mathsf{R} (P, N', D)$, we reason as follows. Since $\mathsf{H} = \{h_u\}$, then $\mathbf{v}' = \mathbf{v}[h_u := 0]$. Now, for the process term, we have $P = \mathsf{age}([\omega : t_1, t_2]^h, \mathbf{v}) = \mathsf{age}([\omega : t_1, t_2]^h, \mathbf{v}')$, since, by safety of clock variable allocation, $h \neq h_u$. For the network, we have $\widehat{N}' = \widehat{N}[k := (\Downarrow, u)^{h_k}]$. But $\mathsf{age}(\widehat{N}, \mathbf{v}) = N$ and $\mathsf{age}((\Downarrow, u)^{h_k}, \mathbf{v}') = (\Downarrow, u)$, so, by safety of clock variable allocation, $\mathsf{age}(\widehat{N}', \mathbf{v}') = N'$. Finally, $D = \widehat{D}$. The result follows.

iii. $\lambda = t \in \mathbb{R}$
A transition $(([\omega : t_1, t_2]^h, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{t} ((\widehat{P}', \widehat{N}', \widehat{D}'), \mathbf{v}')$ must be derived by **TA.2**, so $(\widehat{P}', \widehat{N}' \widehat{D}') = ([\omega : t_1, t_2]^h, \widehat{N}, \widehat{D})$, $\mathbf{v}' = \mathbf{v} + t$, and $\forall\, t' \in \mathbb{R} \mid t' \leq t \,.\, \mathbf{v} + t' \models I([\omega : t_1, t_2]^h, \widehat{N}, \widehat{D})$. By Definition 4.16, $\mathbf{v} + t \models h \leq t_2 \wedge I(\widehat{N})$. Now, we have $P = \mathsf{age}([\omega : t_1, t_2]^h, \mathbf{v}) = [\omega : t_1 \dot{-} \mathbf{v}(h), t_2 \dot{-} \mathbf{v}(h)]$, and, since $\mathbf{v}(h) + t \leq t_2$, then $t \leq t_2 - \mathbf{v}(h)$. Also, we have $N = \mathsf{age}(\widehat{N}, \mathbf{v})$, and, by Lemma B.8, $t \leq \mathsf{tcp}(N)$, so, by **N.4**, we have $N \xrightarrow{t}_{\mathsf{n}} N'$, where $N' = N + t$. Therefore, by **Comp.3**, we derive $(P, N, D) \xrightarrow{t} (P', N', D')$, where $P' = [\omega : t_1 \dot{-} \mathbf{v}(h) \dot{-}$

$t, t_2 \dot{-} \mathbf{v}(h) \dot{-} t]$, and $D' = D$. It is a simple application of the definitions to show that $((\widehat{P}', \widehat{N}', \widehat{D}'), \mathbf{v}')\mathsf{R}(P', N', D')$.

(b) **Case:** $\widehat{\sigma} \equiv ((\widehat{P}\,[> \widehat{Q}, \widehat{N}, \widehat{D}), \mathbf{v})$.
Since $\widehat{\sigma}\mathsf{R}\sigma$, then $\sigma \equiv (P\,[> Q, N, D)$, where $P = \mathsf{age}(\widehat{P}, \mathbf{v})$, $Q = \mathsf{age}(\widehat{Q}, \mathbf{v})$, $N = \mathsf{age}(\widehat{N}, \mathbf{v})$ and $D = \widehat{D}$. There are five sub-cases to consider: four for each of the ways in which an edge can be constructed which justifies a discrete transition by **TA.1**, and one for the justification of a time transition by **TA.2**.

i. **E_Int.1**
In this case, the edge is of the form $(\widehat{P}\,[> \widehat{Q}, \widehat{N}, \widehat{D}) \xrightarrow{\psi, \lambda_{\mathsf{p}}, \mathsf{H}} (\widehat{P}'\,[> \widehat{Q}, \widehat{N}', \widehat{D}')$, with $\lambda = \lambda_{\mathsf{p}} \in A$, $\widehat{P}' \not\equiv\ \checkmark$, $\mathbf{v} \models \psi$, and $\mathbf{v}[\mathsf{H} := 0] \models I(\widehat{P}'\,[> \widehat{Q}, \widehat{N}', \widehat{D}')$. But this edge must be derived from an edge $(\widehat{P}, \widehat{N}, \widehat{D}) \xrightarrow{\psi, \lambda_{\mathsf{p}}, \mathsf{H}} (\widehat{P}', \widehat{N}', \widehat{D}')$. Now, by **TA.1**, this edge justifies a transition $((\widehat{P}, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{\lambda_{\mathsf{p}}} ((\widehat{P}', \widehat{N}', \widehat{D}'), \mathbf{v}[\mathsf{H} := 0])$. So, by i.h., there is a transition $(P, N, D) \xrightarrow{\lambda_{\mathsf{p}}} (P', N', D')$, where there exist $\widehat{P}'' \mathrel{\underline{\leftrightarrow}} \widehat{P}'$ and $P'' \mathrel{\underline{\leftrightarrow}} P'$ such that $((\widehat{P}'', \widehat{N}', \widehat{D}'), \mathbf{v}[\mathsf{H} := 0])\mathsf{R}(P'', N', D')$. Since $\widehat{P}' \not\equiv\ \checkmark$ then, by Definition B.2 and Proposition B.4, we have $P' \not\equiv\ \checkmark$, and so it follows by **Int.1** that there is a transition $(P\,[> Q, N, D) \xrightarrow{\lambda_{\mathsf{p}}} (P'\,[> Q, N', D')$. Since $\widehat{P}'' \mathrel{\underline{\leftrightarrow}} \widehat{P}'$ and $P'' \mathrel{\underline{\leftrightarrow}} P'$, then $\widehat{P}''\,[> \widehat{Q} \mathrel{\underline{\leftrightarrow}} \widehat{P}'\,[> \widehat{Q}$ and $P''\,[> Q \mathrel{\underline{\leftrightarrow}} P'\,[> Q$. To see that $((\widehat{P}''\,[> \widehat{Q}, \widehat{N}', \widehat{D}'), \mathbf{v}[\mathsf{H} := 0])\mathsf{R}(P''\,[> Q, N', D')$, we reason as follows. By construction of R, $P'' = \mathsf{age}(\widehat{P}'', \mathbf{v}[\mathsf{H} := 0])$. Furthermore, by safety of clock variable allocation, $\mathsf{H} \cap \mathsf{iclk}(\widehat{Q}) = \emptyset$, so, by Lemma B.6, $Q = \mathsf{age}(\widehat{Q}, \mathbf{v}[\mathsf{H} := 0])$ and $N' = \mathsf{age}(\widehat{N}', \mathbf{v}[\mathsf{H} := 0])$. The result follows by construction of R.

ii. **E_Int.2** Similar to previous case.

iii. **E_Int.3** Similar to previous case.

iv. **E_Int.4**
In this case the edge is of the form $(\widehat{P}\,[> \widehat{Q}, \widehat{N}, \widehat{D}) \xrightarrow{\psi, \lambda_{\mathsf{n}}, \mathsf{H}} (\widehat{P}\,[> \widehat{Q}, \widehat{N}', \widehat{D})$, where $\lambda = \lambda_{\mathsf{n}} \in A_{\mathsf{n}}$, $\mathbf{v} \models \psi$, and $\mathbf{v}' = \mathbf{v}[\mathsf{H} := 0] \models I(\widehat{P}\,[> \widehat{Q}, \widehat{N}', \widehat{D})$. But the existence of this edge depends upon edges $(\widehat{P}, \widehat{N}, \widehat{D}) \xrightarrow{\psi, \lambda_{\mathsf{n}}, \mathsf{H}} (\widehat{P}, \widehat{N}', \widehat{D})$ and $(\widehat{Q}, \widehat{N}, \widehat{D}) \xrightarrow{\psi, \lambda_{\mathsf{n}}, \mathsf{H}} (\widehat{Q}, \widehat{N}', \widehat{D})$. Now, by **TA.1**, these edges justify transitions $((\widehat{P}, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{\lambda_{\mathsf{n}}} ((\widehat{P}, \widehat{N}', \widehat{D}), \mathbf{v}')$ and $((\widehat{P}, \widehat{N}, \widehat{D}), \mathbf{v}) \xrightarrow{\lambda_{\mathsf{n}}} ((\widehat{P}, \widehat{N}', \widehat{D}), \mathbf{v}')$. So, by i.h. and Lemma B.1, there are transitions $(P, N, D) \xrightarrow{\lambda_{\mathsf{n}}} (P, N', D)$ and $(Q, N, D) \xrightarrow{\lambda_{\mathsf{n}}} (Q, N', D)$, where there exist $\widehat{P}'' \mathrel{\underline{\leftrightarrow}} \widehat{P}$, $P'' \mathrel{\underline{\leftrightarrow}} P$, $\widehat{Q}'' \mathrel{\underline{\leftrightarrow}} \widehat{Q}$, and $Q'' \mathrel{\underline{\leftrightarrow}} Q$ such that $((\widehat{P}'', \widehat{N}', \widehat{D}), \mathbf{v}')\mathsf{R}(P'', N', D)$ and $((\widehat{Q}'', \widehat{N}', \widehat{D}), \mathbf{v}')\mathsf{R}(Q'', N', D)$. Therefore, by **Int.4**, $(P\,[> Q, N, D) \xrightarrow{\lambda_{\mathsf{n}}} (P\,[> Q, N', D)$. Clearly, by i.h. and Definition B.2, $((\widehat{P}''\,[> \widehat{Q}'', \widehat{N}', \widehat{D}), \mathbf{v}')\mathsf{R}(P''\,[> Q'', N', D)$, where $\widehat{P}''\,[> \widehat{Q}'' \mathrel{\underline{\leftrightarrow}} \widehat{P}\,[> \widehat{Q}$ and $P''\,[> Q'' \mathrel{\underline{\leftrightarrow}} P\,[> Q$.

v. $\lambda = t \in \mathbb{R}$

This transition is derived by **TA.2**, where $((\widehat{P}\,[>\widehat{Q},\widehat{N},\widehat{D}),\mathbf{v})\xrightarrow{t}((\widehat{P},\widehat{N},\widehat{D}),\mathbf{v}')$, where $\mathbf{v}'=\mathbf{v}+t$. The transition is possible only if $\forall\, t'\in\mathbb{R}\mid t'\leq t\,.\,\mathbf{v}+t'\models I(\widehat{P}\,[>\widehat{Q},\widehat{N},\widehat{D})$. So, by Definition 4.16, $\mathbf{v}+t\models I(\widehat{P},\widehat{D})\wedge I(\widehat{Q},\widehat{D})\wedge I(\widehat{N})$, and, therefore, by **TA.2**, $((\widehat{P},\widehat{N},\widehat{D}),\mathbf{v})\xrightarrow{t}((\widehat{P},\widehat{N},\widehat{D}),\mathbf{v}')$, and $((\widehat{Q},\widehat{N},\widehat{D}),\mathbf{v})\xrightarrow{t}((\widehat{Q},\widehat{N},\widehat{D}),\mathbf{v}')$. By i.h. and Lemma B.1, $(P,N,D)\xrightarrow{t}(P',N',D)$ and $(Q,N,D)\xrightarrow{t}(Q',N',D)$ and there exist $\widehat{P}''\;\underline{\leftrightarrow}\;\widehat{P}$, $P''\;\underline{\leftrightarrow}\;P'$, $\widehat{Q}''\;\underline{\leftrightarrow}\;\widehat{Q}$ and $Q''\;\underline{\leftrightarrow}\;Q'$ such that $((\widehat{P}'',\widehat{N},\widehat{D}),\mathbf{v}')\mathsf{R}(P',N',D)$ and $((\widehat{Q}'',\widehat{N},\widehat{D}),\mathbf{v}')\mathsf{R}(Q',N',D)$. Therefore, by **Int.4**, $(P\,[>Q,N,D)\xrightarrow{t}(P'\,[>Q',N',D)$, and, by construction of R, $((\widehat{P}''\,[>\widehat{Q}'',\widehat{N},\widehat{D}),\mathbf{v}')\mathsf{R}(P''\,[>Q'',N',D)$, where $\widehat{P}''\,[>\widehat{Q}''\;\underline{\leftrightarrow}\;\widehat{P}\,[>\widehat{Q}$ and $P''\,[>Q''\;\underline{\leftrightarrow}\;P'\,[>Q'$.

(c) **Case:** $\widehat{\sigma}\equiv((\mathrm{rec}\,X.\widehat{P}_1,\widehat{N},\widehat{D}),\mathbf{v})$
So, since $\widehat{\sigma}\mathsf{R}\sigma$, we have $\sigma\equiv(P,N,D)$, where $P=\mathsf{age}(\mathrm{rec}\,X.\widehat{P}_1,\mathbf{v})$, $N=\mathsf{age}(\widehat{N},\mathbf{v})$, and $D=\widehat{D}$. There are two sub-cases to consider:

i. $\lambda\in A_{\mathsf{p}}\cup A_{\mathsf{n}}$:
Then, the transition $((\mathrm{rec}\,X.\widehat{P}_1,\widehat{N},\widehat{D}),\mathbf{v})\xrightarrow{\lambda}((\widehat{P}',\widehat{N}',\widehat{D}'),\mathbf{v}')$ must be derived by **TA.1** from an edge $(\mathrm{rec}\,X.\widehat{P}_1,\widehat{N},\widehat{D})\xrightarrow{\psi,\lambda,\mathsf{H}}(\widehat{P}',\widehat{N}',\widehat{D}')$, where $\mathbf{v}\models\psi$ and $\mathbf{v}'=\mathbf{v}[\mathsf{H}:=0]\models I(\widehat{P}',\widehat{N}',\widehat{D}')$. This edge must be justified by **E_Rec** from an edge $(\widehat{P}_1[\mathrm{rec}\,X.\widehat{P}_1/X],\widehat{N},\widehat{D})\xrightarrow{\psi,\lambda,\mathsf{H}}(\widehat{P}',\widehat{N}',\widehat{D}')$. So, by **TA.1**, there is a transition $((\widehat{P}_1[\mathrm{rec}\,X.\widehat{P}_1/X],\widehat{N},\widehat{D}),\mathbf{v})\xrightarrow{\lambda}((\widehat{P}',\widehat{N}',\widehat{D}'),\mathbf{v}')$. But $\mathsf{age}(\mathrm{rec}\,X.\widehat{P}_1,\mathbf{v})=\mathsf{age}(\widehat{P}_1[\mathrm{rec}\,X.\widehat{P}_1/X],\mathbf{v})$, which is derived by a shorter calculation, and so, by i.h., we have $(P,N,D)\xrightarrow{\lambda}(P',N',D')$ and there exist $\widehat{P}''\;\underline{\leftrightarrow}\;\widehat{P}'$ and $P''\;\underline{\leftrightarrow}\;P'$ such that $((\widehat{P}'',\widehat{N}',\widehat{D}'),\mathbf{v}')\mathsf{R}(P'',N',D')$.

ii. $\lambda=t\in\mathbb{R}$:
Then, the transition $((\mathrm{rec}\,X.\widehat{P}_1,\widehat{N},\widehat{D}),\mathbf{v})\xrightarrow{t}((\widehat{P}',\widehat{N}',\widehat{D}'),\mathbf{v}')$ must be derived by **TA.2**, where $(\widehat{P}',\widehat{N}',\widehat{D}')\equiv(\mathrm{rec}\,X.\widehat{P}_1,\widehat{N},\widehat{D})$, $\mathbf{v}'=\mathbf{v}+t$ and $\forall\, t'\in\mathbb{R}\mid t'\leq t\,.\,\mathbf{v}+t'\models I(\mathrm{rec}\,X.\widehat{P}_1,\widehat{N},\widehat{D})$. But then, by Definition 4.16, we have $\forall\, t'\in\mathbb{R}\mid t'\leq t\,.\,\mathbf{v}+t'\models I(\widehat{P}_1[\mathrm{rec}\,X.\widehat{P}_1/X],\widehat{N},\widehat{D})$, and so, by **TA.2**, there is a transition $((\widehat{P}_1[\mathrm{rec}\,X.\widehat{P}_1/X],\widehat{N},\widehat{D}),\mathbf{v})\xrightarrow{t}((\widehat{P}_1[\mathrm{rec}\,X.\widehat{P}_1/X],\widehat{N},\widehat{D}),\mathbf{v}')$. By i.h., we have $(P,N,D)\xrightarrow{t}(P',N',D')$ and there exist $\widehat{P}''\;\underline{\leftrightarrow}\;\widehat{P}_1[\mathrm{rec}\,X.\widehat{P}_1/X]$ and $P''\;\underline{\leftrightarrow}\;P'$ such that $((\widehat{P}_1[\mathrm{rec}\,X.\widehat{P}_1/X],\widehat{N},\widehat{D}),\mathbf{v}')\,\mathsf{R}(P'',N',D')$. But $\widehat{P}''\;\underline{\leftrightarrow}\;\widehat{P}_1[\mathrm{rec}\,X.\widehat{P}_1/X]\;\underline{\leftrightarrow}\;\mathrm{rec}\,X.\widehat{P}_1$, as required.

$\square$

C. THE *CANDLE* GRAMMAR

C.1 Syntax Notation

The grammar of *CANDLE* is described in an extended Backus-Naur-Form (BNF).

- Italicized words are used to denote syntactic categories (non-terminal symbols), for example:

  module
  formalParameter
  statement

- `Typewriter` font is used for lexical elements of the language (terminal symbols) such as keywords and special symbols, for example:

  `function`
  `>=`
  `every`

- A list of alternative items is written with each alternative occurring on a new line, for example:

  parameterMode ::=
  `in`
  `out`
  `inout`

  Indentation is used to show that a new line is intended as a continuation of the previous item, rather than the beginning of a new alternative.

- [·] denotes an optional item, for example:

  loopStatement ::=
  `loop` [loop*Identifier*] `do` *seqStatement* `end loop`

  is a rule for a `loop` statement, in which a loop*Identifier* may be present but is not required.

- $\{\cdot\}^*$ denotes zero or more occurrences of an item, and $\{\cdot\}^+$ denotes one or more occurrences of an item, for example

module ::=
`module` module*Identifier* `is` {*declSection*}* [*behaviour*] `end module`

is a rule which shows that zero or more occurrences of a *declSection* item may occur in a *module* declaration.

C.2 Lexical Conventions

The lexical conventions of *CANDLE* are standard:

- Whitespace characters are space, tab and newline. Any string starting with two dashes "`--`" and ending with a newline is a *comment*. Multiple-line comments start with the string "`(*`" and end with the string "`*)`".
- A *number* is any sequence of digits.
- An *identifier* is any sequence of characters in the set $\{A-Z, a-z, 0-9, \_\}$ which starts with a letter, excluding the reserved words shown below. All characters in an identifier are significant, and case is significant.
- The *reserved words* are:

`and`	`behaviour`	`channel`	`const`	`do`	`elapse`	
`else`	`elsif`	`end`	`exception`	`exit`	`every`	`function`
`if`	`idle`	`in`	`inout`	`is`	`loop`	`mod`
`module`	`not`	`null`	`or`	`out`	`procedure`	`rcv`
`return`	`select`	`snd`	`then`	`trap`	`type`	`var`

C.3 Modules

program ::=
{*module*}*

module ::=
`module` module*Identifier* `is` {*declSection*}* [*behaviour*] `end module`

C.4 Declarations

declSection ::=
typeDeclSection
constantDeclSection
variableDeclSection
functionDeclSection
procedureDeclSection
channelDeclSection
exceptionDeclSection

C.4.1 Type Declarations

typeDeclSection ::=
`type` *typeDecl* {`;` *typeDecl*}*

typeDecl ::=
type*Identifier*

C.4.2 Constant Declarations

constantDeclSection ::=
`const` *constantDecl* {`;` *constantDecl*}*

constantDecl ::=
constant*Identifier* `:` type*Identifier*

C.4.3 Variable Declarations

variableDeclSection ::=
`var` *variableDecl* {`;` *variableDecl*}*

variableDecl ::=
variable*Identifier* `:` type*Identifier*

C.4.4 Function and Procedure Declarations

functionDeclSection ::=
`function` *functionDecl* {`;` *functionDecl*}*

functionDecl ::=
function*Identifier* `(` [*formalParameter* {`;` *formalParameter*}*] `)` `:` type*Identifier*

procedureDeclSection ::=
`procedure` *procedureDecl* {`;` *procedureDecl*}*

procedureDecl ::=
procedure*Identifier* `(` [*formalParameter* {`;` *formalParameter*}*] `)`

formalParameter ::=
[*parameterMode*] type*Identifier*

parameterMode ::=
`in`
`out`
`inout`

C.4.5 Channel Declarations

channelDeclSection ::=
  `channel` *channelDecl* {`;` *channelDecl*}*

channelDecl ::=
  channel*Identifier* [*channelSpec*]

channelSpec ::=
  `:` `(` *messageSpec* {`,` *messageSpec*}* `)`

messageSpec ::=
  message*Identifier* [`.` type*Identifier*]

C.4.6 Exception Declarations

exceptionDeclSection ::=
  `exception` *exceptionDecl* {`;` *exceptionDecl*}*

exceptionDecl ::=
  exception*Identifier* `:` type*Identifier*

C.5 Expressions

expression ::=
  constantLiteral
  identifier
  `?` exception*Identifier*
  functionCall
  `-` *expression*
  expression `*` *expression*
  expression `/` *expression*
  expression `+` *expression*
  expression `-` *expression*
  expression `mod` *expression*
  expression `=` *expression*
  expression `/=` *expression*
  expression `<` *expression*
  expression `<=` *expression*
  expression `>` *expression*
  expression `>=` *expression*
  `not` *expression*
  expression `and` *expression*
  expression `or` *expression*
  `(` *expression* `)`

The precedence of operators is given below. Operators of equal precedence are shown on the same line. Operators of lower precedence are shown first. All operators are left-associative, except unary minus and logical negation which are non-associative.

```
or
and
not
=   /=   <   <=   >=   >
+   -
*   /   mod
```

constantLiteral ::=
  `uvalue`
  `false`
  `true`
  number

functionCall ::=
  function*Identifier* `(` [*expression* {`,` *expression*}*] `)`

C.6 Behaviour

behaviour ::=
  `behaviour` *statement*

statement ::=
  seqStatement `|` *statement*
  seqStatement

seqStatement ::=
  atomicStatement `;` *seqStatement*
  atomicStatement

atomicStatement ::=
  `null`
  `idle`
  sndStatement
  rcvStatement
  elapseStatement
  assignmentStatement
  procedureCall
  ifStatement
  loopStatement
  everyStatement
  selectStatement
  trapStatement
  exitStatement
  moduleInstantiation

C.6.1 Send statement

sndStatement ::=
`snd` `(` channel*Identifier* `,` message*Identifier* [`.` *expression*] `)`

C.6.2 Receive statement

rcvStatement ::=
`rcv` `(` channel*Identifier* `,` message*Identifier* [`.` variable*Identifier*] `)`

C.6.3 Elapse statement

elapseStatement ::=
`elapse` *expression*

C.6.4 Assignment statement and Procedure Call

assignmentStatement ::=
variable*Identifier* `:=` *expression*

procedureCall ::=
procedure*Identifier* `(` [*expression* {`,` *expression*}*] `)`

C.6.5 If statement

ifStatement ::=
`if` *thenPart* {*elsifPart*}* [*elsePart*] `end if`

thenPart ::=
expression `then` *seqStatement*

elsifPart ::=
`elsif` *expression* `then` *seqStatement*

elsePart ::=
`else` *seqStatement*

C.6.6 Repetition statements

loopStatement ::=
`loop` [loop*Identifier*] `do` *seqStatement* `end loop`

everyStatement ::=
`every` *expression* `do` *seqStatement* `end every`

C.6.7 Select statement

selectStatement ::=
`select` {*selectAlternative*}$^+$ [`in` *seqStatement*] `end select`

selectAlternative ::=
`::` *rcvElapseOrModuleInstantiation* [`;` *seqStatement*]

rcvElapseOrModuleInstantiation ::=
rcvStatement
elapseStatement
moduleInstantiation

C.6.8 Trap statement

trapStatement ::=
`trap` {*trapAlternative*}$^+$ `in` *seqStatement* `end trap`

trapAlternative ::=
`::` exception*Identifier* `=>` *seqStatement*

exitStatement ::=
`exit` exception*Identifier* [`(` *expression* `)`]

C.6.9 Module Instantiation

moduleInstantiation ::=
module*Identifier* [`[` *renaming* {`,` *renaming*}* `]`]

renaming ::=
expression `/` *identifier*

D. THE *SDML* GRAMMAR

D.1 Introduction

SDML is a simple data modelling language which can be used with *CANDLE* for describing the data objects and operations in CAN-based systems.

The notation used in giving the grammar of *SDML* is the same as the notation of Appendix C, as are the lexical conventions, except for the reserved words, which are as follows:

`and`	`any`	`array`	`begin`	`boolean`	`bounds`	`const`
`data`	`do`	`end`	`fi`	`function`	`if`	`in`
`is`	`inout`	`mod`	`not`	`od`	`of`	`or`
`out`	`procedure`	`return`	`skip`	`type`	`uvalue`	`var`

D.2 Data Modules

program ::=
  {*dataModule*}*

dataModule ::=
  `data` dataModule*Identifier* `is` {*declSection*}* `end data`

D.3 Declarations

declSection ::=
  typeDeclSection
  constantDeclSection
  functionDeclSection
  procedureDeclSection

D.3.1 Type Declarations

typeDeclSection ::=
  `type` *typeDecl* {`;` *typeDecl*}*

typeDecl ::=
  type*Identifier* `is` *typeExpr* [`size` *expression*]

typeExpr ::=
  enumerationType
  subrangeType

recordType
arrayType
type*Identifier*

enumerationType ::=
`{` *enumElement* {`,` *enumElement*}* `}`

enumElement ::=
constant*Identifier*
number

subrangeType ::=
expression `..` *expression*

recordType ::=
`{|` *variableDecl* {`;` *variableDecl*}* `|}`

arrayType ::=
`array` *typeExpr* `of` *typeExpr*

D.3.2 Constant Declarations

constantDeclSection ::=
`const` *constantDecl* {`;` *constantDecl*}*

constantDecl ::=
constant*Identifier* `:` type*Identifier* [`is` *expression*]

D.3.3 Function and Procedure Declarations

functionDeclSection ::=
`function` *functionDecl* {`;` *functionDecl*}*

functionDecl ::=
function*Identifier* `(` [*formalParameter* {`;` *formalParameter*}*] `)` `:`
type*Identifier*
`is` [*bounds*] [*variableDeclSection*] *statementPart*

procedureDeclSection ::=
`procedure` *procedureDecl* {`;` *procedureDecl*}*

procedureDecl ::=
procedure*Identifier* `(` [*formalParameter* {`;` *formalParameter*}*] `)`
`is` [*bounds*] [[*variableDeclSection*] *statementPart*]

formalParameter ::=
[*parameterMode*] variable*Identifier* `:` type*Identifier*

parameterMode ::=
  `in`
  `out`
  `inout`

bounds ::=
  `bounds` *bound* `;` *bound*

bound ::=
  expression
  `~`

D.3.4 Variable Declarations

variableDeclSection ::=
  `var` *variableDecl* {`;` *variableDecl*}*

variableDecl ::=
  variable*Identifier* `:` *typeExpr*

D.4 Expressions

assignableExpression ::=
  expression
  `any` type*Identifier*

expression ::=
  designator
  constantLiteral
  functionCall
  `-` *expression*
  expression `*` *expression*
  expression `/` *expression*
  expression `+` *expression*
  expression `-` *expression*
  expression `mod` *expression*
  expression `=` *expression*
  expression `/=` *expression*
  expression `<` *expression*
  expression `<=` *expression*
  expression `>` *expression*
  expression `>=` *expression*
  `not` *expression*
  expression `and` *expression*
  expression `or` *expression*
  `(` *expression* `)`

designator ::=
  variable*Identifier*
  designator `.` field*Identifier*
  designator `[` *expression* `]`

constantLiteral ::=
  `uvalue`
  `false`
  `true`
  number

fieldAssignment ::=
  identifier `=` *expression*

functionCall ::=
  function*Identifier* `(` [*expression* {`,` *expression*}*] `)`

D.5 Statements

statementPart ::=
  `begin` *statement* `end`

statement ::=
  atomicStatement `;` *statement*
  atomicStatement

atomicStatement ::=
  `skip`
  assignmentStatement
  procedureCall
  returnStatement
  ifStatement
  doStatement

D.5.1 Assignment statement and Procedure Call

assignmentStatement ::=
  designator `:=` *assignableExpression*

procedureCall ::=
  procedure*Identifier* `(` [*expression* {`,` *expression*}*] `)`

D.5.2 Return statement

returnStatement ::=
  `return` *expression*

D.5.3 If statement and Repetition

ifStatement ::=
`if` $\{guardedCommand\}^+$ `fi`

doStatement ::=
`do` $\{guardedCommand\}^+$ `od`

guardedCommand ::=
`::` *expression* `=>` *statement*

E. GLOSSARY

Below are the notations used in this dissertation together with the section in which each notation is defined or first appears.

Sets of Numbers

$\mathbb{N}$	Natural numbers	2.2
$\mathbb{N}_\infty$	$\mathbb{N} \cup \{\infty\}$	4.2.1
$\mathbb{Z}$	Integer numbers	2.2
$\mathbb{Z}_\infty$	$\mathbb{Z} \cup \{\infty\}$	2.7.5
$\mathbb{Q}$	Rational numbers	2.2
$\mathbb{R}$	Non-negative real numbers	2.2
$\mathbb{R}_\infty$	$\mathbb{R} \cup \{\infty\}$	2.2

Actions

$\mathbb{R}$	Set of delay actions	2.3.2
t	Delay action of duration t in $\mathbb{R}$	2.3.2
τ	Time-abstracted delay action	2.7.2
A_n	Set of network actions	3.4.2
λ_n	Network action in A_n	3.6.2
λ_nt	Network or delay action in $A_\mathsf{n} \cup \mathbb{R}$	3.6.2
A_p	Set of process actions	3.6.2
λ_p	Process action in A_p	3.6.2
A	Set of discrete actions	2.3.2
a	Discrete action in A	2.3.2
A_τ	$A \cup \{\tau\}$	2.7.2
L	Set of all actions	2.3
λ	Any action in L	2.3

Labelled Transition Systems (LTS)

$\mathcal{S}$	Labelled transition system (LTS)	2.3
Σ	Set of states of a LTS	2.3
$\sigma^\mathcal{I}$	Initial state of a LTS	2.3
σ	State of LTS in Σ	2.3
$\mathbf{p}$	Path $\sigma_0\lambda_0\sigma_1\lambda_1\cdots$ in LTS	2.3
$\mathsf{label}_\mathbf{p}(i)$	Label of i-*th* action in the path $\mathbf{p}$	2.3
$\Xi_\mathcal{S}(\sigma)$	Set of executions (infinite paths) from state σ in LTS $\mathcal{S}$	2.3.2
ξ	Execution in $\Xi_\mathcal{S}$	2.3.2
$\delta_\xi(i)$	Duration of i-*th* action in the execution ξ	2.3.2
$\Delta_\xi(n)$	Time elapsed in ξ from σ_0 to σ_n	2.3.2
$\Xi^\infty_\mathcal{S}(\sigma)$	Set of time divergent executions from σ	2.3.2
$\mathcal{S}_1 \mid \mathcal{S}_2$	Parallel composition of LTS's	2.3.3

Transition Relations

$\longrightarrow$	Transition relation	2.3
$\longrightarrow^\mathrm{n}$	n-ary composition of $\longrightarrow$	2.3
$\longrightarrow^*$	n-ary composition of $\longrightarrow$ for some $n \in \mathbb{N}$	2.3
$\longrightarrow_\mathsf{n}$	Network transition relation	3.4.2
$\longrightarrow_\mathsf{rg}$	Region graph transition relation	2.7.2
$\longrightarrow_\mathsf{sg}$	Simulation graph transition relation	2.7.6

Equivalence Relations

$P \equiv Q$	Syntactic identity	3.3
$P \underline{\leftrightarrow} Q$	Strong bisimulation	3.6.3
$P = Q$	Equality (modulo equational theory)	3.6.4

Clocks

$\mathcal{H}$	Set of clock variables	2.5.2
h	Clock (meta-)variable ranging over $\mathcal{H}$	2.5.2
$\mathbb{R}^{\mathcal{H}}$	Set of clock valuations	2.5.2
$\mathbf{v}$	Clock valuation in $\mathbb{R}^{\mathcal{H}}$	2.5.2
$\mathbf{0}$	Clock valuation with all clock variables equal to 0	2.5.2
$\mathbf{v}[\mathsf{H} := 0]$	Reset of all clocks $h \in \mathsf{H}$	2.5.2
$\mathbf{v} + t$	$\mathbf{v}$ after delay of duration t	2.5.2
$\Psi_{\mathcal{H}}$	Set of clock constraints	2.5.2
χ	Atomic clock constraint	2.5.2
$\mathcal{Z}_{\mathcal{H}}$	Set of conjunctions of atomic clock constraints, convex clock constraints	2.5.2
ζ	Convex clock constraint, $\zeta \in \mathcal{Z}_{\mathcal{H}}$	2.5.2
ψ	Clock constraint, $\psi \in \Psi_{\mathcal{H}}$	2.5.2
$\mathbf{v} \models \psi$	Clock valuation $\mathbf{v}$ satisfies clock constraint ψ	2.5.2
$[\![\psi]\!]$	Characteristic set of ψ	2.5.2
tt, ff	True and False clock constraints	2.5.2

Timed Automata (TA)

$\mathcal{A}$	Timed Automaton	2.5
Q	Set of locations	2.5.4
$q^{\mathcal{I}}$	Initial location	2.5.4
q	Location in Q	2.5.4
E	Set of edges	2.5.4
e	Edge in E	2.5.4
$\mathsf{guard}(e)$	Guard of edge e	2.5.4
$\mathsf{label}(e)$	Action label of edge e	2.5.4
$\mathsf{reset}(e)$	Reset clocks of edge e	2.5.4
I	Invariant function	2.5.4
$c_{max}(\mathcal{A})$	Largest constant value in guard or invariant of $\mathcal{A}$	2.5.4
$\mathcal{A}_1 \mid \mathcal{A}_2$	Parallel composition of TA	2.5.6
$\mathcal{T}[\![\mathcal{A}]\!]$	LTS derived from TA $\mathcal{A}$	2.5.5
$(q, \mathbf{v})$	State in the LTS of a TA	2.5.5
$(q, \mathbf{v}) + t$	Alternative to $(q, \mathbf{v} + t)$	2.5.5
$(q, \mathbf{v})[\mathsf{H} := 0]$	Alternative to $(q, \mathbf{v}[\mathsf{H} := 0])$	2.5.5
$\mathsf{RG}(\mathcal{A})$	Region graph of the TA $\mathcal{A}$	2.7.2
$\mathsf{SG}(\mathcal{A})$	Simulation graph of the TA $\mathcal{A}$	2.7.6
$\mathsf{AG}(\mathcal{A})$	Activity graph of the TA $\mathcal{A}$	5.2.2

Polyhedra

ψ	$\mathcal{H}$-polyhedron, $\psi \in \Psi_{\mathcal{H}}$	2.7.4
ζ	Convex $\mathcal{H}$-polyhedron, $\zeta \in \mathcal{Z}_{\mathcal{H}}$	2.7.4
$\mathsf{suc}_e(\psi)$	Polyhedron denoting successors of ψ via TA edge e	2.7.4
$\mathsf{suc}_\tau^q(\psi)$	Polyhedron denoting successors of ψ as time passes at TA location q	2.7.4
$\psi_1 \cap \psi_2$	Intersection of polyhedra	2.7.4
$\psi_1 \cup \psi_2$	Union of polyhedra	2.7.4
$\psi_1 \setminus \psi_2$	Difference of polyhedra	2.7.4
$\psi_1 \subseteq \psi_2$	Inclusion of polyhedra	2.7.4
$\psi_1 \sqcup \psi_2$	Convex hull of polyhedra	2.7.4
$\overline{\psi}$	Complement of polyhedron	2.7.4
$\nearrow \psi$	Forward projection	2.7.4
$\psi[\mathsf{H} := 0]$	Reset projection	2.7.4
$\mathsf{close}_c(\zeta)$	c-closure	2.7.4

Difference Bound Matrices (DBM)

$(c, \prec)$	Bound in $\mathbb{Z}_\infty \times \{<, \leq\}$	2.7.5
M	Difference Bound Matrix	2.7.5
$M_{i,j}$	Bound at row i, column j in DBM M	2.7.5
$[\![M]\!]$	Set of clock valuations denoted by DBM M	2.7.5
$M^\emptyset$	Empty DBM, i.e. $[\![M^\emptyset]\!] = \emptyset$	2.7.5
$\mathbf{U}$	Universal DBM, i.e. $[\![\mathbf{U}]\!] = \mathbb{R}^{\mathcal{H}}$	2.7.5
$\mathsf{cf}\ M$	Canonical form of DBM M	2.7.5

Data Environment

Var	Set of data variables	3.3.1
x	Data variable in Var	3.3.1
V	Set of data values	3.3.1
v	Data value in V	3.3.1
$\mathsf{type}(x)$	Type of data variable x	3.3.1
$\mathit{Valuation}$	Set of data valuations, $\mathit{Var} \to V$	3.3.1
$\mathsf{val}(x)$	Value of data variable x	3.3.1
Ω	Set of data operation names	3.3.1
ω	Data operation name in Ω	3.3.1
$\mathsf{operation}(\omega)$	Operation which interprets the operation name ω	3.3.1
Γ	Set of data predicate names	3.3.1
γ	Data predicate name in Γ	3.3.1
$\mathsf{predicate}(\gamma)$	Predicate which interprets the predicate name γ	3.3.1
$\mathit{DataEnv}_{\mathit{Var},\Omega,\Gamma}$	Set of data environments over Var, Ω and Γ	3.3.1
D	Data environment in $\mathit{DataEnv}$	3.3.1
$D.x$	Value of data variable x given by D	3.3.1
$D[x := v]$	Update value of x to become v	3.3.1
$D \xrightarrow{\omega}_{\mathsf{d}} D'$	Data environment transformation by operation ω	3.3.1
ID	Identity data operation	3.3.1
$\mathit{true}, \mathit{false}$	True and False data predicates	3.3.1

Channels

I	Set of message identifiers	3.4.1
M	Set of messages, $M \subseteq I \times V$	3.4.1
$i.v$	Message with identifier i and data value v	3.4.1
$m \prec m'$	Message m has higher priority than m'	3.4.1
δ	Transmission latency function, derived functions are $\delta^{\mathsf{lb}}, \delta^{\mathsf{ub}}, \delta^{\mathsf{lB}}$, and δ^{uB}	3.4.1
$Status_{M,\delta}$	Set of transmission statuses	3.4.1
s	Transmission status in $Status$	3.4.1
$\downarrow$	Free status	3.4.1
$\overset{t_1,t_2}{\rightsquigarrow} m$	Pre-acceptance phase of message transmission	3.4.1
$\uparrow m$	Acceptance point in message transmission	3.4.1
$m \overset{t_1,t_2}{\rightsquigarrow}$	Post-acceptance phase of message transmission	3.4.1
$Queue_{M,\prec}$	Set of message queues	3.4.1
u	Message queue in $Queue$	3.4.1
$\langle\rangle$	Empty message queue	3.4.1
$m{:}u$	Message queue with highest priority message m	3.4.1
$u \hookleftarrow m$	Prioritised insertion of m into u	3.4.1
$Channel_I$	Set of channels over I	3.4.1
η	Channel in $Channel$	3.4.1

Networks

K	Set of channel identifiers	3.4.1
k	Channel identifier in K	3.4.1
$Network_{K,I}$	Set of networks, $K \to Channel_I$	3.4.1
$\widehat{Network}_{K,I}$	Set of clocked networks, $K \to Channel_I \times \mathcal{H}$	4.2.1
N_k	Channel k in network N	3.4.1
$\widehat{N}_k$	Channel k in clocked network $\widehat{N}$	4.2.1
$N[k := \eta]$	Update of channel k in network N	3.4.2
$\mathsf{tcp}(N)$	Maximum time progress for network N	3.4.2
$N + t$	State of network N after delay of duration t	3.4.2

Process Constructions

$k!i.x$	Send broadcast message	3.5
$k?i.x$	Receive broadcast message	3.5
$[\omega : t_1, t_2]$	Time bounded computation	3.5
$\gamma \rightarrow P$	Data guard	3.5
$P \,;\, P$	Sequential composition	3.5
$P + P$	Non-deterministic choice	3.5
$P \;[> P$	Interrupt	3.5
$P \mid P$	Parallel composition	3.5
$\mathrm{rec}\, X.P$	Recursion	3.5
idle	$[\mathit{ID} : \infty]$	3.5
null	$[\mathit{ID} : 0]$	3.5
Proc^+	Set of process terms	3.5
Proc	Set of closed, guarded process terms	3.5
P, Q	Process terms in Proc^+ or Proc	3.5
β	Basic term in Proc^+ or Proc	3.5
$\widehat{\mathit{Proc}}$	Set of clocked, closed, guarded process terms	4.2.2
$\widehat{P}, \widehat{Q}$	Clocked process terms in $\widehat{\mathit{Proc}}$	4.2.2
$\widehat{\beta}$	Clocked basic term in $\widehat{\mathit{Proc}}$	4.2.2

Nets

$\mathcal{R}$	A net (W, Θ, W^I)	4.4.1
W	Set of places	4.4.1
w	Place in W	4.4.1
Θ	Set of transitions	4.4.1
θ	Transition in Θ	4.4.1
W^I	Set of initial places	4.4.1
${}^\bullet\theta$	Trigger place of θ	4.4.1
${}^\circ\theta$	Vulnerable places to θ	4.4.1
$\theta^\bullet$	Target places of θ	4.4.1
$\alpha\theta$	Attribute of θ	4.4.1
θ_w	Transition triggered by place w	4.4.1

Minimised Automata (MA)

$\mathcal{A}$	Minimised deterministic finite state automaton	5.3.1
Q	Set of states of a MA	5.3.1
Q_i	Set of states at layer i in a MA	5.3.1
Q^-	$Q \setminus Q_k$, for a k-layer MA	5.3.1
A	Alphabet of a MA	5.3.1
E	Transition relation of a MA	5.3.1
T, F	Accepting and rejecting final states in a MA	5.3.1
$\mathbf{a}$	A string $a_0, a_1, \ldots, a_{n-1}$	5.3.1
$\mathcal{L}_{\mathcal{A}}(q)$	Language of $\mathcal{A}$ from state q	5.3.1
$\mathcal{L}(\mathcal{A})$	$\mathcal{L}_{\mathcal{A}}(q_0)$	5.3.1

BIBLIOGRAPHY

[AAB+99] P. Abdulla, A. Annichini, S. Bensalem, A. Bouajjani, P. Habermehl, and Y. Lakhnech. Verification of infinite-state systems by combining abstraction and reachability analysis. In N. Halbwachs and D. Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*. Springer Verlag, 1999.

[ABBL98] L. Aceto, P. Bouyer, A. Burgueño, and K. Larsen. The power of reachability testing for timed automata. Technical Report RS-98-48, BRICS,Department of Computer Science,University of Aarhus, December 1998.

[ABK+97] E. Asarin, M. Bozga, A. Kerbrat, O. Maler, A. Pnueli, and A. Rasse. Data structures for the verification of timed automata. In O. Maler, editor, *Proceedings of 1st International Workshop on Hybrid and Real-Time Systems (HART'97)*, volume 1201 of *Lecture Notes in Computer Science*, pages 346–360. Springer Verlag, March 1997.

[ABL96] J.-R. Abrial, E. Börger, and H. Langmaack, editors. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, volume 1165 of *Lecture Notes in Computer Science*. Springer Verlag, 1996.

[ABL98] L. Aceto, A. Burgueño, and K. Larsen. Model checking via reachability testing for timed automata. In B. Steffen, editor, *Proceedings of 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *Lecture Notes in Computer Science*, pages 263–280. Springer Verlag, 1998.

[Abr96] J.-R. Abrial. *The B Book – Assigning Programs to Meanings.* Cambridge University Press, 1996.

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proceedings of 5th IEEE Symposium on Logic in Computer Science*, pages 414–425. IEEE Computer Society Press, June 1990.

[ACD+92] R. Alur, C Courcoubetis, D. Dill, N. Halbwachs, and H. Wong-Toi. An implementation of three algorithms for timing verification based on automata emptiness. In *Proceedings of the 13th IEEE Real-Time Systems Symposium*, pages 157–166, 1992.

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1):2 – 34, May 1993.

[ACdP97] L. Andriantsiferana, J.-P. Courtiat, R. de Oliveira, and L. Picci. An experiment in using RT-LOTOS for the formal specification and verification of a distributed scheduling algorithm in a nuclear power plant monitoring system. In *Proceedings of IFIP Joint International Conference on Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE-PSTV'98), Osaka, Japan.* Chapman and Hall, November 1997.

[ACH+92] R. Alur, C. Courcoubetis, N. Halbwachs, D. Dill, and H. Wong-Toi. Minimization of timed transition systems. In Cleaveland [Cle92], pages 340–354.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.

[AD90] R. Alur and D. Dill. Automata for modelling real-time systems. In M. Paterson, editor, *Proceedings of 17th International Colloquium on Automata, Languages and Programming (ICALP'90)*, volume 443 of *Lecture Notes in Computer Science*, pages 322–335. Springer Verlag, 1990.

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–236, 1994. Preliminary version appears in Proceedings of 17th ICALP, 1990, LNCS 443.

[AD96] R. Alur and D. Dill. Automata-theoretic verification of real-time systems. In *Formal Methods for Real-Time Computing*, Trends in Software Series, pages 55–82. John Wiley & Sons Publishers, 1996.

[AFV99] L. Aceto, W. Fokkink, and C. Verhoef. Structural operational semantics. Technical Report RS-99-30, BRICS,Department of Computer Science,Aarhus University, 1999.

[AH91] R. Alur and T. Henzinger. Logics and models of real time: A survey. In J. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of REX Workshop, Real-Time:Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 74–106. Springer Verlag, 1991.

[AH93] R. Alur and T. Henzinger. Real-time logics: complexity and expressiveness. *Information and Computation*, 104(1):35–77, 1993. Preliminary version appears in the Proceedings of 5th LICS, 1990.

[AHP96] R. Alur, T. Henzinger, and Pei-Hsin Ho. Automatic symbolic verification of embedded systems. *IEEE Transactions on Software Engineering*, 22(3):181–201, March 1996.

[AHS95] R. Alur, T. Henzinger, and E. Sontag, editors. *Proceedings of DIMACS/SYCON Workshop on Verification and Control of Hybrid Systems (Hybrid Systems III)*, volume 1066 of *Lecture Notes in Computer Science*. Springer Verlag, October 1995.

[AIKY95] R. Alur, A. Itai, R. Kurshan, and M. Yannakakis. Timing verification by successive approximation. *Information and Computation*, 118:142–157, 1995.

[AK95] R. Alur and R. Kurshan. Timing analysis in COSPAN. In Alur et al. [AHS95], pages 220–231.

[ALST98] M. Ammerlaan, R. Lutje-Spelberg, and W. Toetenel. XTG – an engineering approach to modelling and analysis of real-time systems. In *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, pages 88–97. IEEE Computer Society Press, 1998.

[Alu91] R. Alur. *Techniques for Automatic Verification of Real-time Systems*. PhD thesis, Stanford University, 1991.

[AMP98] E. Asarin, O. Maler, and A. Pnueli. On discretization of delays in timed automata and digital circuits. In R. de Simone and D. Sangiorgi, editors, *Proceedings of the 9th International Conference of Concurrency Theory (CONCUR'98)*, volume 1466 of *Lecture Notes in Computer Science*, pages 470–484. Springer Verlag, 1998.

[Bal96] F. Balarin. Approximate reachability analysis of timed automata. In *Proceedings of 17th IEEE Real Time Systems Symposium*, pages 52–61. IEEE Computer Society Press, 1996.

[Bar96] J. Barnes. *High Integrity Development with SPARK Ada*. Addison Wesley, 1996.

[BB91] J. Baeten and J. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2):142–188, 1991.

[BB97] A. Benzekri and J.-M. Bruel. Controller Area Network: A formal case study. In *Proceedings of IFAC Workshop on Factory Communications*. IFAC, 1997.

[BCM+92] J.R. Burch, E.M. Clarke, K.L McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BD91] B. Berthomieu and M. Diaz. Modeling and verification of time-dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17(3):259–273, 1991.

[BDM+98] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: a model-checking tool for real-time systems. In Hu and Vardi [HV98], pages 546–550.

[BDS94] J. Bryans, J. Davies, and S. Schneider. Real-time CSP and ET-LOTOS. Technical report, Oxford PRG, 1994.

[Bel57] R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[Ber98a] G. Berry. *The Esterel v5 Language Primer, Version 5.10, release 2.0.* Centre de Mathématiques Appliquées, Ecole des Mines and INRIA, 2004 Route des Lucioles, 06565 Sophia-Antipolis, 1998.

[Ber98b] G. Berry. *The Esterel v5 Manual, Version 5.10, release 2.0.* Centre de Mathématiques Appliquées, Ecole des Mines and INRIA, 2004 Route des Lucioles, 06565 Sophia-Antipolis, 1998.

[BFG99a] M. Bozga, J.-C. Fernandez, and L. Ghirvu. State space reduction based on live variables analysis. In A. Cortesi and G. File, editors, *Proceedings of 6th Static Analysis Symposium*, volume 1694 of *Lecture Notes in Computer Science*, pages 164–178. Springer Verlag, 1999.

[BFG+99b] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, and L. Mounier. IF: An intermediate representation and validation environment for timed asynchronous systems. In J. Wing, J. Woodcock, and J. Davies, editors, *Proceedings of FM'99, Toulouse, France*, volume 1709 of *Lecture Notes in Computer Science*, pages 307–327. Springer Verlag, 1999.

[BFH+92] A. Bouajjani, J-C. Fernandez, N. Halbwachs, P. Raymond, and C. Ratel. Minimal state graph generation. *Science of Computer Programming*, 18:247–269, 1992.

[BFK+98] H. Bowman, G. Faconti, J.-P. Katoen, D. Latella, and M. Massink. Automatic verification of a lip synchronization algorithm using UPPAAL. In *Proceedings of the 3rd International Workshop on Formal Methods for Industrial Critical Systems*, 1998.

[BG92] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19:87–152, 1992.

[BHKR94] S. Bradley, W. Henderson, D. Kendall, and A. Robson. Designing and implementing correct real-time systems. In H. Langmaack, W.-P. de Roever, and J. Vytopil, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems FTRTFT '94, Lübeck, Lecture Notes in Computer Science 863*, pages 228–246. Springer Verlag, September 1994.

[BHKR95] S. Bradley, W. Henderson, D. Kendall, and A. Robson. Validation, verification and implementation of timed protocols using AORTA. In P. Dembinski, editor, *Proceedings of the Fifteenth International Symposium on Protocol Specification, Testing and Verification*, pages 205–220. Chapman and Hall, June 1995.

[BHKR01] S. Bradley, W. Henderson, D. Kendall, and A. Robson. A formal design language for real-time systems with data. *Science of Computer Programming*, 40(1):3–29, May 2001.

[BHR84] S. Brookes, C. Hoare, and A. Roscoe. A theory of communicating sequential processes. *Journal of the ACM*, 31(3):560–599, 1984.

[BJLY98] J. Bengtsson, B. Jonsson, J. Lilius, and W. Yi. Partial order reductions for timed systems. In R. de Simone and D. Sangiorgi, editors, *Proceedings of the 9th International Conference of Concurrency Theory (CONCUR'98)*, volume 1466 of *Lecture Notes in Computer Science*, pages 485–500. Springer Verlag, 1998.

[BK84] J. Bergstra and J. Klop. The algebra of recursively defined processes and the algebra of regular processes. In J. Paredaens, editor, *Proceedings of 11th International Colloquium on Automata, Languages and Programming (ICALP'84),Antwerp,Belgium*, volume 172 of *Lecture Notes in Computer Science*, pages 82–95. Springer Verlag, 1984.

[BL91] T. Bolognesi and F. Lucidi. Timed process algebras with urgent interactions and a unique powerful binary operator. In J. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of REX Workshop, Real-Time:Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 124–148. Springer Verlag, 1991.

[BLL+95] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL – a tool suite for automatic verification of real-time systems. In Alur et al. [AHS95], pages 232–243.

[BLL+98] J. Bengtsson, K. Larsen, F. Larsson, P. Pettersson, W. Yi, and C. Weise. New generation of Uppaal. In Hu and Vardi [HV98].

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. InVeSt: A tool for the verification of invariants. In A. Hu and M. Vardi, editors, *Proceedings of the 10th International Conference on Computer*

Aided Verification (CAV'98), volume 1427 of *Lecture Notes in Computer Science*, pages 505–510. Springer Verlag, 1998.

[BLP+99] G. Behrmann, K. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient timed reachability analysis using clock difference diagrams. In N. Halbwachs and D. Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 341–353. Springer Verlag, 1999.

[BLSTV99] A. Burns, R. Lutje-Spelberg, H. Toetenel, and T. Vink. Modeling and verification using XTG and PMC. In *Proceedings of 5th Annual Conference of Advanced School for Computing and Imaging, Heijen, The Netherlands*, June 1999.

[BM00] O. Bournez and O. Maler. On the representation of timed polyhedra. In U. Montanari, J. Rolim, and E. Wetzl, editors, *Proceedings of 27th International Colloquium on Automata, Languages and Programming (ICALP'00)*, volume 1853 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[BMPY97] M. Bozga, O. Maler, A. Pnueli, and S. Yovine. Some progress in the symbolic verification of timed automata. In O. Grumberg, editor, *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 179–190, Haifa, Israel, June 1997. Springer Verlag.

[Bos91] R. Bosch GmbH. CAN specification version 2.0, September 1991.

[Bow99] J. Bowen. Animating the semantics of VERILOG using Prolog. Technical Report UNU/IIST Report No. 176, International Institute for Software Technology, United Nations University (UNU/IIST), 1999.

[Boz98] M. Bozga. SMI: An open toolbox for symbolic protocol verification. Technical report, VERIMAG, April 1998.

[BR00] J. Baeten and M. Reniers. Termination in timed process algebra. Technical Report CSR 00-13, Department of Computing Science, Eindhoven University of Technology, 2000.

[Bra95] S. Bradley. *An Implementable Formal Language for Hard Real-Time Systems*. PhD thesis, University of Northumbria, 1995.

[BRB90] K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a BDD package. In *Proceedings of the 27th ACM/IEEE Conference on Design Automation*, pages 40–45. IEEE Computer Society Press, 1990.

[BRS93] G. Berry, S. Ramesh, and R. Shyamasundar. Communicating reactive processes. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 85–98, Charleston, South Carolina, 1993.

[Bry86] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 8(C-35):677–691, 1986.

[Bry92] R. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[BTY97] A. Bouajjani, S. Tripakis, and S. Yovine. On-the-fly symbolic model-checking for real-time systems. In *Proceedings of 18th IEEE Real Time Systems Symposium*, pages 25–34. IEEE Computer Society Press, 1997.

[BV94] B. Bloom and F. Vaandrager. SOS rule formats for parameterized and state-bearing processes. Unpublished manuscript, July 1994.

[BV95] J. Baeten and C. Verhoef. Concrete process algebra. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Volume IV, Syntactical Methods*, pages 149–268. Oxford University Press, 1995.

[BV97] J. Baeten and J. Vereijken. Discrete-time process algebra with empty process. Technical Report CSR 97-05, Department of Computing Science, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands, 1997.

[BVW94] O. Bernholtz, M. Vardi, and P. Wolper. An automata-theoretic approach to branching-time model checking. In *Proceedings of the 6th International Conference on Computer Aided Verification (CAV'94)*, volume 818 of *Lecture Notes in Computer Science*, pages 142–155. Springer Verlag, 1994.

[BW90] J. Baeten and W. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science 18. Cambridge University Press, 1990.

[BW01] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 3rd edition, 2001.

[CA91] S. Chamberlain and P. Amer. Broadcast channels in Estelle. *IEEE Transactions on Computers*, 40(4):423–436, 1991.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of 4th ACM Symposium on Principles of Programming Languages (POPL'77)*, 1977.

[CDH+00] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.

[CDI99] F. Corradini, D. D'Ortenzio, and P. Inverardi. On the relationships among four timed process algebras. *Fundamenta Informaticae*, 34:1–19, 1999.

[CdO95] J.-P. Courtiat and R. de Oliveira. A reachability analysis of RT-LOTOS specifications. In G. von Bochmann, R. Dssouli, and O. Rafiq, editors, *Proceedings of International Conference on Formal Description Techniques VIII (FORTE'95), Montreal, Canada*, pages 117–124. Chapman and Hall, October 1995.

[CE81] E. Clarke and E. Emerson. Design and synthesis of synchronisation skeletons using branching time temporal logic. In *Proceedings of Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Verlag, 1981.

[Čěr92] K. Čěrans. Decidability of bisimulation equivalence for parallel timer processes. In *Proceedings of the 4th International Conference on Computer Aided Verification (CAV'92)*, volume 663 of *Lecture Notes in Computer Science*, pages 302–315. Springer Verlag, 1992.

[CES86] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.

[CGL94] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[CHR91] Z. Chaochen, C. Hoare, and A. Ravn. A calculus of durations. *Information Processing Letters*, 40(5):269–276, December 1991.

[CiA99] CiA. *Proceedings of the 6th International CAN Conference*. CAN in Automation, 1999.

[CK96] E. Clarke and R. Kurshan. Computer-aided verification. *IEEE Spectrum*, 33(6):61–67, 1996.

[Cle92] W. Cleaveland, editor. *Proceedings of the 3rd International Conference of Concurrency Theory (CONCUR'92)*. Springer Verlag, 1992.

[CO92] R. Cardell-Oliver. *The Formal Verification of Hard Real-Time Systems*. PhD thesis, University of Cambridge, 1992.

[COG98] R. Cardell-Oliver and T. Glover. A practical and complete algorithm for testing real-time systems. In *Proceedings of International Conference on Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT'98)*, volume 1486 of *Lecture Notes in Computer Science*, pages 251–261. Springer Verlag, September 1998.

[Cor96] J. Corbett. Timing analysis of Ada tasking programs. *IEEE Transactions on Software Engineering*, 22(7):461–483, July 1996.

[CPS93] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics-based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

[CVGH98] M. Colnaric, D. Verber, R. Gumzej, and W. Halang. Implementation of hard real-time embedded control systems. *Journal of Real-Time Systems*, 14:293–310, 1998.

[DAC98] M. Dwyer, G. Avrunin, and J. Corbett. Property specification patterns for finite-state verification. In M. Ardis, editor, *Proceedings of 2nd Workshop on Formal Methods in Software Practice*, pages 7–15. ACM Press, March 1998.

[Dav93] J. Davies. *Specification and Proof in Real-Time CSP*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1993.

[Daw98a] C. Daws. *Méthodes d'analyse de systèmes temporisés: de la théorie á la pratique*. PhD thesis, Institut National Polytechnique de Grenoble, 1998. (In French).

[Daw98b] C. Daws. Optikron: a tool suite for enhancing model-checking of real-time systems. In *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 542–545. Springer Verlag, 1998.

[DB96] P. D'Argenio and E. Brinksma. A calculus for timed automata. In B. Jonsson and J. Parrow, editors, *Proceedings of the 4th International Conference on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'96)*, volume 1135 of *Lecture Notes in Computer Science*, pages 110–129. Springer Verlag, 1996.

[dBHdRR91] J. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg, editors. *Proceedings of REX Workshop, Real-Time:Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.

[DF95] J. Dingel and T. Filkorn. Model checking for infinite state systems using data abstraction, assumption-commitment style reasoning and theorem proving. In P. Wolper, editor, *Proceedings of the 7th International Conference on Computer Aided Verification (CAV'95)*, volume 939 of *Lecture Notes in Computer Science*, pages 54–69. Springer Verlag, 1995.

[DGKK98] D. Dams, R. Gerth, B. Knaack, and R. Kuiper. Partial-order reduction techniques for real-time model-checking. *Formal Aspects of Computing*, 10:469–482, 1998.

[Die01] H. Dierks. PLC-automata: a new class of implementable real-time automata. *Theoretical Computer Science*, 253:61–93, 2001.

[Dij76] E. Dijkstra. *A Discipline of Programming.* Prentice Hall, Englewood Cliffs, N.J., 1976.

[Dil89] D. Dill. Timing assumptions and verification of finite state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer Verlag, 1989.

[DIN89] DIN. Profibus standard — Deutsche Industrie Norm (DIN 19245) (2 parts). Beuth-Verlag, Berlin, 1989.

[DJS92] J. Davies, D. Jackson, and S. Schneider. Broadcast communication for real-time processes. In J. Vytopil, editor, *Proceedings of International Conference on Formal Techniques in Real-time and Fault-tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, pages 149–170. Springer Verlag, January 1992.

[DKRT97] P. D'Argenio, J.-P. Katoen, T. Ruys, and J. Tretmans. The bounded retransmission protocol must be on time! In E. Brinksma, editor, *Proceedings of 3rd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 416–431. Springer Verlag, 1997.

[DOTY95] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In Alur et al. [AHS95], pages 208–219.

[DOY94] C. Daws, A. Olivero, and S. Yovine. Verifying ET-LOTOS programs with KRONOS. In *Proceedings of International Conference on Formal Description Techniques VII (FORTE'94)*, pages 227–242. Chapman and Hall, 1994.

[DPC98] M. Dwyer, C. Păsăreanu, and J. Corbett. Translating Ada programs for model checking: A tutorial. Technical Report 98-12, Department of Computing and Information Sciences, Kansas State University, 1998.

[DT98] C. Daws and S. Tripakis. Model checking of real-time reachability properties using abstractions. In Steffen [Ste98], pages 313–329.

[DW99] M. Dickhöfer and T. Wilke. Timed alternating tree automata: The automata-theoretic solution to the TCTL model checking problem. In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP'99)*, volume 1643 of *Lecture Notes in Computer Science*, pages 281–290. Springer Verlag, 1999.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with Kronos. In *Proceedings of 16th IEEE Real Time Systems Symposium*, pages 66–75. IEEE Computer Society Press, December 1995.

[DY96] C. Daws and S. Yovine. Reducing the number of clock variables of timed automata. In *Proceedings of 17th IEEE Real Time Systems Symposium*, pages 73–81. IEEE Computer Society Press, December 1996.

[Ech91] Echelon Corp. Enhanced media access control with Echelon's LonTalk protocol. LonWorks Engineering Bulletin, August 1991.

[EFM99] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proceedings of 14th IEEE Symposium on Logic in Computer Science (LICS'99)*, pages 352–359. IEEE Computer Society Press, 1999.

[EH86] E. Emerson and J. Halpern. 'Sometimes' and 'Not Never' revisited: on branching versus linear temporal logic. *Journal of the ACM*, 33(1):151–178, 1986.

[Eme90] E. Emerson. Temporal and modal logic. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 995–1072. Elsevier Science, 1990.

[Eme91] E. Emerson. Real-time and the Mu-calculus. In J. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of REX Workshop, Real-Time:Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 176–194. Springer Verlag, June 1991.

[EMSS90] E. Emerson, A. Mok, A. Sistla, and J. Srinivasan. Quantitative temporal reasoning. In R. Kurshan and E. Clarke, editors, *Proceedings of the 2nd International Conference on Computer Aided Verification (CAV'90)*, volume 531 of *Lecture Notes in Computer Science*, pages 136–145. Springer Verlag, 1990.

[FGK+96] J. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP a protocol validation and verification toolbox. In *Proceedings of the 8th International Conference* *on Computer Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer Verlag, 1996.

[FKFV99] R. Fraer, G. Kamhi, L. Fix, and M. Vardi. Evaluating semi-exhaustive verification techniques for bug hunting. *Electronic Notes in Theoretical Computer Science*, 23(2), 1999.

[FM91] J. Fernandez and L. Mounier. "On the Fly" verification of behavioural equivalences and preorders. In K. Larsen, editor, *Proceedings of the 3rd International Conference on Computer Aided Verification (CAV'91)*, volume 575 of *Lecture Notes in Computer Science*, pages 238–250. Springer Verlag, July 1991.

[GA98] M. Ganai and A. Aziz. Efficient coverage directed state space search. In *Proceedings of International Workshop on Logic Synthesis, Lake Tahoe, CA*, 1998.

[Gar92] H. Garavel. *Compilation et Vérification de Programmes LOTOS*. PhD thesis, Institut National Polytechnique de Grenoble, July 1992. (In French).

[Gar98] H. Garavel. OPEN/CÆSAR: An open software architecture for verification, simulation and testing. In B. Steffen, editor, *Proceedings of 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *Lecture Notes in Computer Science*, pages 68–84. Springer Verlag, March 1998.

[GdV99] J. Geldenhuys and P. de Villiers. Runtime efficient state compaction in SPIN. In *Proceedings of 5th International SPIN Workshop, Trento, Italy*. Springer Verlag, July 1999.

[Geh84] N. Gehani. Broadcasting sequential processes (BSP). *IEEE Transactions on Software Engineering*, 10(4):343–351, July 1984.

[GGZ95] F. Gagnon, J.-C. Grégoire, and D. Zampunieris. Sharing trees for "on-the-fly" verification. In *Proceedings of International Conference on Formal Description Techniques VIII (FORTE'95), Montreal, Canada*. IEEE Computer Society Press, 1995.

[GHP95] P. Godefroid, G. Holzmann, and D. Pirottin. State-space caching revisited. *Formal Methods and System Design*, 7(3):1–15, November 1995.

[GL94] O. Grumberg and D. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.

[God96] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*, volume 1032 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1996.

[God97] P. Godefroid. Model checking for programming languages using Verisoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 174–186. ACM Press, 1997.

[Gol96] U. Golze. *VLSI Chip Design with the Hardware Description Language VERILOG*. Springer Verlag, Berlin, 1996.

[GP94] J. Groote and A. Ponse. Process algebra with guards: Combining Hoare logic with process algebra. *Formal Aspects of Computing*, 6:115–164, 1994.

[GPVW95] R. Gerth, D. Peled, M. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In P. Dembinski, editor, *Proceedings of the Fifteenth International Symposium on Protocol Specification, Testing and Verification*, pages 1–18. Chapman Hall, 1995.

[Gré96] J.-Ch. Grégoire. State space compression in SPIN with GE-sets. In *Proceedings of 2nd SPIN Workshop, Rutgers University, New Jersey, USA*, August 1996.

[Hal93] W. Halang. Contemporary computers considered inappropriate for real-time control. *Control Engineering Practice*, 1(4):613–621, 1993.

[Har87] D. Harel. Statecharts: A visual approach to complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[HDZ00] J. Hatcliff, M. Dwyer, and H. Zheng. Slicing software for model construction. *Higher Order and Symbolic Computation*, 13(4):315–353, 2000.

[Hen96] T. Henzinger. The theory of hybrid automata. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science (LICS'96)*, pages 278–292. IEEE Computer Society Press, 1996.

[Her98] C. Hernalsteen. *Specification, Validation and Verification of Real-Time Systems in ET-LOTOS*. PhD thesis, Université Libre de Bruxelles, August 1998.

[HHWT97] T. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *Springer International Journal of Software Tools for Technology Transfer*, 1(1–2):110–122, October 1997.

[HKPV95] T Henzinger, P. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? In *Proceedings of 27th Annual Symposium on Theory of Computing*, pages 373–382, 1995.

[HKV96] T. Henzinger, O. Kupferman, and M. Vardi. A space-efficient on-the-fly algorithm for real-time model checking. In *Proceedings of the 7th International Conference of Concurrency Theory (CONCUR'96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 514–529. Springer Verlag, 1996.

[HLP90] E. Harel, O. Lichtenstein, and A. Pnueli. Explicit clock temporal logic. In *Proceedings of 5th IEEE Symposium on Logic in Computer Science*, pages 402–413. IEEE Computer Society Press, 1990.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *IEEE Trans. on Software Engineering*, 18(9):785–793, September 1992.

[HMP92] T. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *19th International Colloquium on Automata, Languages and Programming*, volume 623 of *Lecture Notes in Computer Science*, pages 545–558. Springer Verlag, 1992.

[HNSY94] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, 1994.

[Hoa69] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–583, 1969.

[Hoa85] C. Hoare. *Communicating Sequential Processes.* Englewood Cliffs. Prentice Hall International, 1985.

[Hol85] G. Holzmann. Tracing protocols. *AT&T Technical Journal*, 64(12):2413–2434, 1985.

[Hol90] G. Holzmann. Algorithms for automated protocol validation. *AT&T Technical Journal*, 69(1):32–44, 1990.

[Hol94] U. Holmer. *On broadcast and real time in process calculi.* PhD thesis, University of Goteborg, 1994.

[Hol95] G. Holzmann. An analysis of bitstate hashing. In P. Dembinski, editor, *Proceedings of the Fifteenth International Symposium on Protocol Specification, Testing and Verification*, pages 301–314. Chapman & Hall, June 1995.

[Hol96] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 22(6):279–295, June 1996.

[Hol97] G. Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In *Proceedings of 3rd SPIN Workshop, Twente University, Enschede, Netherlands*, April 1997.

[Hoo91] J. Hooman. Compositional verification of real-time systems using extended Hoare triples. In J. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of REX Workshop, Real-Time:Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 252–290. Springer Verlag, 1991.

[Hoo96] J. Hooman. Assertional specification and verification. In M. Joseph, editor, *Real-time Systems: Specification, Verification and Analysis*, pages 97–146. Prentice Hall International, 1996.

[HP94] G. Holzmann and D. Peled. An improvement in formal verification. In *Proceedings of International Conference on Formal Description Techniques VII (FORTE'94)*, pages 197–211. Chapman and Hall, 1994.

[HP99] G. Holzmann and A. Puri. A minimized automaton representation of reachable states. *Software Tools for Technology Transfer*, 3(1), 1999.

[HP00] K. Havelund and T. Pressburger. Model checking Java programs using Java Pathfinder. *Springer International Journal of Software Tools for Technology Transfer*, 2(4), April 2000.

[HQR98] T. Henzinger, S. Qadeer, and S. Rajamani. You Assume, We Guarantee: Methodology and case studies. In A. Hu and M. Vardi, editors, *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 440–451. Springer Verlag, 1998.

[HS91] W. Halang and A. Stoyenko. *Constructing Predictable Real Time Systems*. Kluwer Academic Publishers, 1991.

[HS99] G. Holzmann and M. Smith. A practical method for verifying event-driven software. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, pages 597–607, May 1999.

[HSLL97] K. Havelund, A. Skou, K. Larsen, and K. Lund. Formal modeling and analysis of an audio/video protocol: An industrial case study using UPPAAL. In *Proceedings of 18th IEEE Real Time Systems Symposium*, pages 2–13. IEEE Computer Society Press, December 1997.

[HU79] J. Hopcroft and J. Ullman. *Introduction to automata theory, languages and computation*. Addison Wesley, 1979.

[Huc99] F. Huch. Verification of Erlang programs using abstract interpretation and model checking. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, pages 261–272, September 1999.

[Hun99] T. Hune. Modelling a real-time language. In *Proceedings of the 4th Workshop on Formal Methods for Industrial Critical Systems (FMICS'99), Trento,Italy*, June 1999.

[HV98] A. Hu and M. Vardi, editors. *Proceedings of the 10th International Conference on Computer Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[HW98] P.-A. Hsiung and F. Wang. A State Graph Manipulator tool for real-time system specification and verification. In *Proceedings of 5th International Conference on Real-Time Computing Systems and Applications (RTCSA'98)*. IEEE Computer Society Press, October 1998.

[HWT95] P.-H. Ho and H. Wong-Toi. Automated analysis of an audio control protocol. In P. Wolper, editor, *Proceedings of the 7th International Conference on Computer Aided Verification (CAV'95)*, volume 939 of *Lecture Notes in Computer Science*, pages 381–394. Springer Verlag, 1995.

[HWT96] T. Henzinger and H. Wong-Toi. Using HYTECH to synthesize control parameters for a steam boiler. In Abrial et al. [ABL96], pages 265–282.

[IKL$^+$00] T. Iversen, K. Kristoffersen, K. Larsen, M. Laursen, R. Madsen, S. Mortensen, P. Pettersson, and C. Thomasen. Model-checking real-time control programs. In *Proceedings of 12th EuroMicro Conference on Real-Time Systems*. IEEE Computer Society Press, June 2000.

[ISO88a] ISO. Estelle – a formal description technique based on an extended state transition model. International Standard 9074, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Geneva, 1988.

[ISO88b] ISO. LOTOS — a formal description technique based on the temporal ordering of observational behaviour. International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Geneva, September 1988.

[ISO92] ISO/DIS 11898: Road Vehicles – interchange of digital information – Controller Area Network (CAN) for high speed communication, 1992.

[ISO96] ISO. Iso/iec tr 14252:1996, information technology – guide to the posix open system environment (ose), December 1996.

[ISO98] ISO. ISO/IEC JTC1/SC21 WG7 Enhancements to LOTOS, May 1998.

[JJ91] C. Jard and T. Jeron. Bounded-memory algorithms for verification on-the-fly. In *Proceedings of the 3rd International Conference on Computer Aided Verification (CAV'91)*, volume 575 of *Lecture Notes in Computer Science*, pages 189–196. Springer Verlag, 1991.

[JLM88] F. Jahanian, R. Lee, and A. Mok. Semantics of Modecharts in real time logic. In *Proceedings of 21st Hawaii International Conference on System Science*, pages 479–489. IEEE Computer Society Press, 1988.

[JM87] F. Jahanian and A. Mok. A graph-theoretic approach for timing analysis and its implementation. *IEEE Transactions on Computers*, C-36(8):961–975, 1987.

[JM95] M. Jourdan and F. Maraninchi. Static timing analysis of real-time systems. *ACM SIGPLAN Notices: Workshop on Languages, Compilers and Tools for Real-Time Systems*, 30(11):79–87, June 1995.

[JO99] L. Johansson and J. Ohlsson. QWIK, a concept for short distance data communication in vehicles and similar applications. HiSafe Development Research Report Version 1.0, 1999.

[Jon90] C. Jones. *Systematic Software Development Using VDM*. Prentice Hall International, second edition, 1990.

[Jos91] M. Joseph. Problems, promises and performance: Some questions for real-time system specification. In J. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of REX Workshop, Real-Time:Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 315–324. Springer Verlag, 1991.

[JW96] D. Jackson and J. Wing. Lightweight formal methods. *IEEE Computer*, 29(4):21–22, April 1996.

[Kel76] R. Keller. Formal verification of parallel programs. *Communications of the ACM*, 19(7):371–384, 1976.

[KG93] H. Kopetz and G. Gruensteidl. TTP — a time-triggered protocol for fault-tolerant real-time systems. In *Proceedings of the 23rd IEEE International Symposium on Fault-Tolerant Computing (FTCS'93), Toulouse, France*, pages 524–532. IEEE Computer Society Press, 1993.

[KLL+97] K. Kristoffersen, F. Laroussinie, K. Larsen, P. Pettersson, and W. Yi. A compositional proof of a real-time mutual exclusion protocol. In *Proceedings of 7th International Joint Conference on the Theory and Practice of Software Development (TAPSOFT'97)*, volume 1214 of *Lecture Notes in Computer Science*, pages 565–579. Springer Verlag, April 1997.

[Kop97] H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[Koy91] R. Koymans. (Real) time: A philosophical perspective. In J. de Bakker, C. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Proceedings of REX Workshop, Real-Time:Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 353–370. Springer Verlag, 1991.

[KP92] Y. Kesten and A. Pnueli. Timed and hybrid statecharts and their textual representation. In J. Vytopil, editor, *Formal Techniques in Real Time and Fault Tolerant Systems*, volume 571 of *Lecture Notes in Computer Science*, pages 591–620. Springer Verlag, January 1992.

[KP98] Y. Kesten and A. Pnueli. Modularization and abstraction: The keys to practical formal verification. In *Proceedings of Mathematical Foundations of Computer Science (MFCS'98)*, volume 1450 of *Lecture Notes in Computer Science*, pages 54–71. Springer Verlag, 1998.

[KS97] C. Krishna and K. Shin. *Real-Time Systems*. The McGraw Hill Companies, Inc., 1997.

[Kur94] R. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton University Press, 1994.

[LAB+98] J. Lind-Nielsen, H. Andersen, G. Behrmann, H. Hulgaard, K. Kristoffersen, and K. Larsen. Verification of large state/event systems using compositionality and dependency analysis. In B. Steffen, editor, *Proceedings of 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *Lecture Notes in Computer Science*, pages 201–216. Springer Verlag, March 1998.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, March 1977.

[Lam80] L. Lamport. 'Sometimes' is sometimes 'Not Never' – on the temporal logic of programs. In *Proceedings of 7th ACM Symposium on Principles of Programming Languages*, pages 174–185, 1980.

[Lap90] J.-C. Laprie. Dependability: Basic concepts and associated terminology. Technical Report PDCS 31, Predictably Dependable Computing Systems (ESPRIT BRA Project 3092), 1990.

[LBGG94] I. Lee, P. Brémond-Grégoire, and R. Gerber. A process algebraic approach to the specification and analysis of resource-bound real-time systems. *Proceedings of IEEE*, pages 158–171, January 1994.

[LGS$^+$95] C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1–35, 1995.

[Liu96] Z. Liu. Specification and verification in DC. In M. Joseph, editor, *Real-time Systems: Specification, Verification and Analysis*, pages 182–228. Prentice Hall International, 1996.

[LK99] M. Livani and J. Kaiser. A total ordering scheme for real-time multicasts in CAN. In *Proceedings of the 24th IFAC/IFIP Workshop on Real-Time Programming*, pages 173–178. IFAC, 1999.

[LKJ99] M. Livani, J. Kaiser, and W. Jia. Scheduling hard and soft real-time communication in the controller area network (CAN). *Control Engineering Practice*, 7(12):1512–1523, December 1999.

[LL95] F. Laroussinie and K. Larsen. Compositional model checking of real-time systems. In *Proceedings of the 6th International Conference of Concurrency Theory (CONCUR'95)*, volume 965 of *Lecture Notes in Computer Science*, pages 27–41. Springer Verlag, 1995.

[LL98] F. Laroussinie and K. Larsen. CMC: A tool for compositional model checking of real-time systems. In *Proceedings of IFIP Joint International Conference on Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE-PSTV'98)*, pages 439–456. Kluwer Academic Publishers, November 1998.

[LLPY97] K. Larsen, F. Larsson, P. Pettersson, and W. Yi. Efficient verification of real-time systems: Compact data structure and state-space reduction. In *Proceedings of 18th IEEE Real Time Systems Symposium*, pages 14–24. IEEE Computer Society Press, December 1997.

[LMW95] S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings of 16th IEEE Real-time Systems Symposium*, pages 298–307, 1995.

[LP97] H. Lönn and P. Pettersson. Formal verification of a TDMA protocol startup mechanism. In *Proceedings of the Pacific Rim International Symposium on Fault-Tolerant Systems*, pages 235–242, December 1997.

[LPY95] K. Larsen, P. Pettersson, and W. Yi. Model-checking for real-time systems. In *Proceedings of Fundamentals of Computation*

Theory, volume 965 of *Lecture Notes in Computer Science*, pages 62–88. Springer Verlag, August 1995.

[LPY97] K. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Springer International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, October 1997.

[LPY98] M. Lindahl, P. Pettersson, and W. Yi. Formal design and analysis of a gear-box controller: an industrial case study using UPPAAL. In Steffen [Ste98], pages 281–297.

[LS98] M. Lowry and M. Subramaniam. Abstraction for analytic verification of concurrent software systems. In *Proceedings of Symposium on Abstraction, Reformulation and Approximation*, Pacific Grove, California, May 1998.

[LSTA98] R. Lutje-Spelberg, H. Toetenel, and M. Ammerlan. Partition refinement in real-time model checking. In *Proceedings of International Conference on Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT'98)*, volume 1486 of *Lecture Notes in Computer Science*, pages 143–157. Springer Verlag, September 1998.

[LV95] N. Lynch and F. Vaandrager. Forward and backward simulations: Part II: Timed Systems. *Information and Computation*, 128(1):1–25, July 1995.

[LWYP98] K. Larsen, C. Weise, W. Yi, and J. Pearson. Clock difference diagrams. Technical Report Nr 98/99, DoCs, Uppsala University, August 1998. ISSN 0283-0574.

[LY92] D. Lee and M. Yannakakis. On-line minimization of transition systems. In *Proceedings of 24th ACM Symposium on Theory of Computing*, pages 264–274, 1992.

[Mar92] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *Proceedings of the 3rd International Conference of Concurrency Theory (CONCUR'92)*, volume 630 of *Lecture Notes in Computer Science*. Springer Verlag, 1992.

[MB83] M. Measche and B. Berthomieu. Time Petri nets for analyzing and verifying time dependent protocols. In H. Rudin and C. West, editors, *Protocol Specification, Verification and Testing III*. IFIP, North Holland, 1983.

[McM92] K. McMillan. *Symbolic Model Checking: An approach to the state explosion problem, CMU-CS-92-131.* PhD thesis, School of Computer Science, Carnegie Mellon University, May 1992.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[ML98] J. Møller and J. Lichtenberg. Difference decision diagrams. Master's Thesis, Department of Information Technology, Technical University of Denmark, August 1998.

[MLAH99] J. Møller, J. Lichtenberg, H. Andersen, and H. Hulgaard. On the symbolic verification of timed systems. Technical Report IT-TR-1999-024, Department of Information Technology, Technical University of Denmark, 1999.

[MP92] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification.* Springer Verlag, New York, 1992.

[MT90] F. Moller and C. Tofts. A temporal calculus of communicating systems. In J. Baeten and J. Klop, editors, *Proceedings of the 1st International Conference on Concurrency Theory (CONCUR'90)*, volume 458 of *Lecture Notes in Computer Science*, pages 401–415. Springer Verlag, 1990.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[Mye79] G. Myers. *The Art of Software Testing.* John Wiley & Sons Ltd., 1979.

[Nat97] National Aeronautics and Space Administration. Formal methods specification and analysis guidebook for the verification of software and computer systems. Volume II: A practitioner's companion. NASA-GB-001-97, May 1997.

[Nic92] X. Nicollin. *ATP: Une algèbre pour la spécification des systèmes temps réel.* PhD thesis, Institut National Polytechnique de Grenoble, France, May 1992. (In French).

[NS91] X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In de Bakker et al. [dBHdRR91], pages 526–548.

[NS94] X. Nicollin and J. Sifakis. The algebra of timed processes, ATP: Theory and application. *Information and Computation*, 114:131–178, 1994.

[NSY91] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to timed graphs and hybrid systems. In de Bakker et al. [dBHdRR91], pages 549–572.

[NSY92] X. Nicollin, J. Sifakis, and S. Yovine. Compiling real-time specifications into extended automata. *IEEE Transactions on Software Engineering*, 18(9):794 – 804, 1992.

[Oli94] A. Olivero. *Modélisation et Analyse de Systèmes Temporisés et Hybrides.* PhD thesis, Institut National Polytechnique de Grenoble, France, September 1994. (In French).

[Ost86] J. Ostroff. Real-time computer control of discrete event systems modelled by extended state machines: A temporal logic approach. Technical Report 8618, Systems Control Group, Dept. of Electrical Engineering, Univ. of Toronto, September 1986. Revised January 1987.

[Pag96] F. Pagani. Partial orders and verification of real time systems. In B. Jonsson and J. Parrow, editors, *Formal Techniques in Real Time and Fault Tolerant Systems*, volume 1135 of *Lecture Notes in Computer Science*, pages 327–346. Springer Verlag, 1996.

[Pag97] F. Pagani. *Ordres Partiels Pour la Vérification de Systèmes Temps Réel*. PhD thesis, Institut National Polytechnique de Grenoble, 1997. (In French).

[Pel92] D. Peled. Sometimes "some" is as good as "all". In Cleaveland [Cle92], pages 192–206.

[Pet99] P. Pettersson. *Modelling and Analysis of Timed Systems: Theory and Practice*. PhD thesis, Department of Computer Science, Uppsala University, February 1999.

[PJF96] S. Peyton Jones and S. Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM Symposium on the Principles of Programming Languages (POPL'96)*, St. Petersburg Beach, Florida, USA, January 1996.

[Plo81] G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI-FN-19, Aarhus University, 1981.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proceedings of 18th IEEE Symposium on Foundations of Computer Science*, pages 46–77. IEEE Computer Society Press, 1977.

[Pnu85] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *Proceedings of 12th International Colloquium on Automata, Languages and Programming (ICALP'85)*, pages 15–32. Springer Verlag, 1985.

[Pnu99] A. Pnueli. From requirements to implementations: A seamless development process for embedded systems. In N. Halbwachs and D. Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*. Springer Verlag, 1999.

[Pra95] K. Prasad. A calculus of broadcasting systems. *Science of Computer Programming*, 25, 1995.

[Pra96] K. Prasad. Broadcasting in time. In *COORDINATION*, volume 1061 of *Lecture Notes in Computer Science*. Springer Verlag, April 1996.

[QS81] J. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proceedings of 5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351. Springer Verlag, 1981.

[RGR98] L. Rodrigues, M. Guimarães, and J. Rufino. Fault-tolerant clock synchronization in CAN. In *Proceedings of 19th IEEE Real Time Systems Symposium*. IEEE Computer Society Press, 1998.

[Rok93] T. Rokicki. *Representing and Modeling Digital Circuits*. PhD thesis, Stanford University, 1993.

[RSS95] S. Rajan, N. Shankar, and M. Srivas. An integration of model checking with automated proof checking. In P. Wolper, editor, *Proceedings of the 7th International Conference on Computer Aided Verification (CAV'95)*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97. Springer Verlag, 1995.

[RVA+98] J. Rufino, P. Veríssimo, G. Arroz, C. Almeida, and L. Rodrigues. Fault-tolerant broadcasts in CAN. In *Digest of Papers, 28th IEEE International Symposium on Fault-Tolerant Computing Systems, Munich, Germany*. IEEE Computer Society Press, 1998.

[SAE92] SAE. Controller Area Network CAN, an in-vehicle serial communication protocol. In *SAE Handbook*, pages 20.341 – 20.355. SAE Press, 1992.

[SBLS99] K. Stahl, K. Baukus, Y. Lakhnech, and M. Steffen. Divide, abstract and model-check. In *Proceedings of 5th International SPIN Workshop, Trento,Italy*, July 1999.

[Sch86] D. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.

[Sch95] S. Schneider. An operational semantics for timed CSP. *Information and Computation*, 116(2):193–213, 1995.

[Sif77] J. Sifakis. Use of Petri nets for performance evaluation. In H. Beilner and E. Gelenbe, editors, *Measuring, Modelling and Evaluating Computer Systems*, pages 75–93. North Holland, 1977.

[Sig98] M Sighireanu. *LOTOS NT User Manual and Report*. INRIA, 1998.

[Sok96] O. Sokolsky. *Efficient Graph-Based Algorithms for Model Checking in the Modal Mu-Calculus*. PhD thesis, State University of New York at Stony Brook, May 1996.

[Spi88] M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics*, volume 3 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1988.

[SS95] O. Sokolsky and S. Smolka. Local model checking for real time systems. In P. Wolper, editor, *Proceedings of the 7th International Conference on Computer Aided Verification (CAV'95)*, volume 939 of *Lecture Notes in Computer Science*, pages 211–224. Springer Verlag, 1995.

[SS98] D. Schmidt and B. Steffen. Program analysis as model checking of abstract interpretations. In G. Levi, editor, *Proceedings of 5th Static Analysis Symposium*, volume 1503 of *Lecture Notes in Computer Science*, pages 351–380. Springer Verlag, 1998.

[SS99] H. Saïdi and N. Shankar. Abstract and model check while you prove. In N. Halbwachs and D. Peled, editors, *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)*, volume 1633 of *Lecture Notes in Computer Science*, pages 443–454. Springer Verlag, July 1999.

[Sta88] J. Stankovic. *Tutorial: Hard Real-Time Systems*, chapter Real-Time Computing Systems: The Next Generation, pages 14–37. IEEE Computer Society Press, 1988.

[Ste97] U. Stern. *Algorithmic Techniques in Verification by Explicit State Enumeration.* PhD thesis, Department of Computer Science, Stanford University, 1997.

[Ste98] B. Steffen, editor. *Proceedings of 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *Lecture Notes in Computer Science*. Springer Verlag, 1998.

[Str99] K. Strehl. Interval diagrams: Increasing efficiency of symbolic real-time verification. In *Proceedings of International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, pages 188–191. IEEE Computer Society Press, 1999.

[SVD97] J. Springintveld, F. Vaandrager, and P. D'Argenio. Testing timed automata. Technical Report CSI-R9712, Computing Science Institute, University of Nijmegen, August 1997.

[TAKB96] S. Taşıran, R. Alur, R. Kurshan, and R. Brayton. Verifying abstractions of timed systems. In *Proceedings of the 7th International Conference of Concurrency Theory (CONCUR'96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 546–562. Springer Verlag, 1996.

[Tan92] A. Tanenbaum. *Modern Operating Systems.* Prentice Hall International, 1992.

[TBK95] H. Touati, R. Brayton, and R. Kurshan. Testing language containment for ω-automata using BDDs. *Information and Computation*, 118:101–109, 1995.

[TBW95] K. Tindell, A. Burns, and A. Wellings. Analysis of hard real-time communications. *Real Time Systems*, 9:147–171, 1995.

[TC96] S. Tripakis and C. Courcoubetis. Extending Promela and Spin for real time. In T. Margaria and B. Steffen, editors, *Proceedings of 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lecture Notes in Computer Science*, pages 329–348. Springer Verlag, 1996.

[Tho90] W. Thomas. Automata on infinite objects. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 133–164. Elsevier Science, 1990.

[THW94] K. Tindell, H. Hansson, and A. Wellings. Analysing real-time communications: Controller Area Network (CAN). In *Proceedings of 15th IEEE Real Time Systems Symposium*, pages 259–263. IEEE Computer Society Press, December 1994.

[Tör98] M. Törngren. Fundamentals of implementing real-time control applications in distributed computer systems. *Journal of Real-Time Systems*, 14:219–250, 1998.

[Tri98] S. Tripakis. *The Formal Analysis of Timed Systems in Practice*. PhD thesis, Université Joseph Fourier, Grenoble, December 1998.

[TY96] S. Tripakis and S. Yovine. Analysis of timed systems based on time-abstracting bisimulations. In *Proceedings of the 8th International Conference on Computer Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 232–243. Springer Verlag, 1996.

[TY98] S. Tripakis and S. Yovine. Verification of the Fast Reservation Protocol with Delayed Transmission using KRONOS. In *Proceedings of 4th IEEE Real Time Technology and Applications Symposium (RTAS'98), Denver, Colorado*, pages 165–170. IEEE Computer Society Press, June 1998.

[UK94] B. Upender and P. Koopman. Communication protocols for embedded systems. *Embedded Systems Programming*, 7(11):46–58, 1994.

[Val93] A. Valmari. On-the-fly verification with stubborn sets. In C. Courcoubetis, editor, *Proceedings of the 5th International Conference on Computer Aided Verification (CAV'93)*, volume 697 of *Lecture Notes in Computer Science*, pages 397–408. Springer Verlag, 1993.

[Var96] M. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata*, volume 1043 of *Lecture Notes in Computer Science*, pages 238–266. Springer Verlag, 1996.

[Ver97a] J. Vereijken. *Discrete-Time Process Algebra*. PhD thesis, Eindhoven University of Technology, 1997.

[Ver97b] P. Veríssimo. Real-time communication. In S. Mullender, editor, *Distributed Systems (2nd edition)*, pages 447–486. Addison Wesley, 1997.

[vG90] R. van Glabeek. The linear time – branching time spectrum. In J. Baeten and J. Klop, editors, *Proceedings of the 1st International Conference on Concurrency Theory (CONCUR'90)*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer Verlag, 1990.

[vG93] R. van Glabeek. The linear time – branching time spectrum II: the semantics of sequential processes with silent moves. In E. Best, editor, *Proceedings of the 4th International Conference of Concurrency Theory (CONCUR'93)*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81. Springer Verlag, 1993.

[Vis96] W. Visser. Memory efficient state storage in SPIN. In *Proceedings of 2nd SPIN Workshop, Rutgers University, New Jersey, USA*, August 1996.

[VRM97] P. Veríssimo, J. Rufino, and L. Ming. How hard is hard real-time communication on field-buses? In *Digest of Papers, 27th IEEE International Symposium on Fault-Tolerant Computing Systems, Washington, USA*. IEEE Computer Society Press, June 1997.

[VW86] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proceedings of 1st IEEE Symposium on Logic in Computer Science*, pages 332–344. IEEE Computer Society Press, 1986.

[VW94] M. Vardi and P. Wolper. Reasoning about infinite computations. *Information and Computation*, 115(1):1–37, November 1994.

[Wan00] F. Wang. Efficient data structure for fully symbolic verification of real-time software systems. In S. Graf and M. Schwartzbach, editors, *Proceedings of 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000)*, volume 1785 of *Lecture Notes in Computer Science*. Springer Verlag, 2000.

[WH98a] F. Wang and P.-A. Hsiung. Automatic verification on the large. In *Proceedings 3rd IEEE High Assurance Systems Engineering Symposium (HASE'98)*. IEEE Computer Society Press, November 1998.

[WH98b] F. Wang and P.-A. Hsiung. Iterative refinement and condensation for state graph construction. Technical Report TR-IIS-98-009, Institute of Information Science, Academia Sinica, Taipei, Taiwan, 1998.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages.* Foundations of Computing Series. MIT Press, 1993.

[Wol97] P. Wolper. The meaning of "formal": from weak to strong formal methods. *Springer International Journal of Software Tools for Technology Transfer*, 1(1–2):6–8, 1997.

[Won95] H. Wong-Toi. *Symbolic Approximations for Verifying Real-Time Systems.* PhD thesis, Department of Computer Science, Stanford University, March 1995.

[WTD94] H. Wong-Toi and D. Dill. Approximations for verifying timing properties. In T. Rus and C. Rattray, editors, *Theories and Experiences for Real-time System Development.* World Scientific Publishing, 1994.

[Yi90] W. Yi. Real-time behaviour of asynchronous agents. In J. Baeten and J. Klop, editors, *Proceedings of the 1st International Conference on Concurrency Theory (CONCUR'90)*, volume 458 of *Lecture Notes in Computer Science*, pages 502–520. Springer Verlag, 1990.

[YL93] M. Yannakakis and D. Lee. An efficient algorithm for minimizing real-time transition systems. In C. Courcoubetis, editor, *Proceedings of the 5th International Conference on Computer Aided Verification (CAV'93)*, volume 697 of *Lecture Notes in Computer Science*, pages 210–224. Springer Verlag, 1993.

[YMW93] J. Yang, A. Mok, and F. Wang. Symbolic model checking for event-driven real-time systems. In *Proceedings of 14th IEEE Real-Time Systems Symposium*, pages 23–32. IEEE Computer Society Press, December 1993.

[Yov93] S. Yovine. *Méthodes et Outils pour la Vérification Symbolique de Systèmes Temporisés.* PhD thesis, Institut National Polytechnique de Grenoble, May 1993. (In French).

[Yov97] S. Yovine. Model checking timed automata. In G. Rozenberg and F. Vaandrager, editors, *Embedded Systems, Papers from the European Educational Forum School on Embedded Systems, Veldhoven, The Netherlands*, volume 1494 of *Lecture Notes in Computer Science*, pages 114–152. Springer Verlag, 1997.

[YPD94] W. Yi, P. Pettersson, and M. Daniels. Automatic verification of real-time systems by constraint solving. In *Proceedings of 7th International Conference on Formal Description Techniques*, 1994.

[YS96] T. Yoneda and B.-H. Schlingloff. Efficient verification of parallel real-time systems. *Journal of Formal Methods in System Design*, 1996.

[YSAA97] J. Yuan, J. Shen, J. Abraham, and A. Aziz. On combining formal and informal verification. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 376–387, 1997.

[Zam97] D. Zampuniéris. *The Sharing Tree Data Structure*. PhD thesis, Facultés Universitaires Notre-Dame de la Paix, Namur, Belgium, May 1997.